REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 23, 2007	Final Report May 2002 - April 2007

4. TITLE AND SUBTITLE
Scalable Technology for a New Generation of Collaboration Applications

5. FUNDING NUMBERS
F49620-02-1-0233

6. AUTHOR(S)
Birman, Ken; Demers, Al; Gehrke, Johannes; Marzullo, Keith; Voelker, Geoff

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
Ithaca, NY 14853

University of California, San Diego
La Jolla, CA 92093-0934

8. PERFORMING ORGANIZATION REPORT NUMBER
41131

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR
Suite 325, Room 3112
875 Randolph Street
Arlington, VA 22203-1768

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-SR-AR-TR-07-0433

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; distribution is Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

Our MURI effort emerged from dialog between the AFRL team developing software for the Joint Battlespace Infosphere (JBI) and university researchers at Cornell and elsewhere. It became clear that to be successful, the JBI needed to break completely new ground in offering publish-subscribe capabilities on a scale never previously attempted, and do so with guarantees of security, reliability and predictable performance of a sort impossible for existing commercial products. This report details the effort, processes, and resulting technologies developed.

14. SUBJECT TERMS
Publish-subscribe, dependent failures, availability management, survivable systems, distributed databases

15. NUMBER OF PAGES
225

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Scalable Technology for a New Generation of Collaborative Applications

MURI Grant Final Report
AFOSR F49620-02-1-0233
May 2002 – April 2007

1. Principal Investigators

Ken Birman (Cornell University)
Al Demers (Cornell University)
Johannes Gehrke (Cornell University)
Keith Marzullo (University of California at San Diego)
Geoffrey M. Voelker (University of California at San Diego)

2. Research Objectives

Our MURI effort emerged from dialog between the AFRL team developing software for the Joint Battlespace Infosphere (JBI) and university researchers at Cornell and elsewhere. It became clear that to be successful, the JBI needed to break completely new ground in offering publish-subscribe capabilities on a scale never previously attempted, and do so with guarantees of security, reliability and predictable performance of a sort impossible for existing commercial products. The AFRL JBI team reacted to this unique challenge by evaluating a number of candidate "core" technologies for distributed systems that map in clear ways to the technical needs of the JBI:

- Group communication (multicast) systems and commercial publish-subscribe systems have a direct correspondence to the communications needs of the JBI.
- The JBI repository will be a database that can be updated and queried in real-time, and there are thus parallels between repository functionality and commercial products in the database area.
- Because many JBI information sources provide data streams or periodic data bursts (indeed, few are likely to be static), there is a strong connection between JBI reporting functionality and the technology of distributed data mining and triggered actions.
- The JBI will use off-the-shelf Web Services technologies wherever possible, thus the degree of match between the JBI and such systems must also be better understood.

It quickly became clear that no existing commercial product is adequate for the full spectrum of JBI requirements. While a number of prototypes have been constructed for various components of the JBI using readily available technologies, these integrate poorly and lack the distributed computing functionality and scalability properties required of the real system. On the basis of commercial experience, any JBI system built in this manner will be fragile and easily disrupted under stress, and omit

important functionality (such as integration of the publish-subscribe aspects of the JBI and the repository) that are lacking in existing commercial offerings. Accordingly, AFRL encouraged the research community to look both at the technical needs of the JBI and its sibling systems in the Navy, Army and Marines, and also to work towards the elaboration of a scientific basis for reasoning about systems such as the JBI - a science some are dubbing "Infospherics" because of its applicability to the Infosphere.

Particularly important are demonstrations of scalable solutions which can be counted upon to remain stable under stress and to offer predictable performance and reliability even when the system is disrupted by an adversary while performing mission-critical tasks. These needs extend from the publish-subscribe functionality per-se to other aspects, such as querying data residing in networks of sensors. As the JBI is adapted for use by other services, some of these aspects may require urgent attention. For example, the Navy is considering the JBI as a platform for future USW sensor applications, but these are primarily data mining problems, not publish-subscribe applications. Understanding how to make such solutions coexist in a single platform is vital for the JBI to emerge into the desired role for the Air Force and its sibling military services.

At Cornell and the University of San Diego, our team came together to pursue common interests at the intersection of large-scale distributed computing systems, Web Services and similar emerging architectures, data mining, and distributed database systems. Our group spans the technology areas needed by the JBI and is united by a shared interest in similar kinds of technologies (particularly probabilistic approaches with good scaling properties), similar application models (relating to publish-subscribe, data mining, and other forms of communication patterns best described as forms of query evaluation), and extensive expertise in experimental evaluation of the technologies we develop.

Birman and van Renesse jointly head the Spinglass group, which is a broad effort developing a new family of reliable, scalable protocols for communication, monitoring, management and control in large distributed systems. Spinglass exploits "multipeer" communication to achieve scalable, probabilistically reliable solutions to component problems that arise in a variety of setting. These components can then be used as tools for enabling direct, live collaboration between participants who may be spread across a global network and using platforms with differing communications capabilities.

Birman and his group focused on scalable group communication systems that provide critical properties such as data replication, distributed coordination, and automated reaction to faults. They developed systems that provide these properties and scale publish-subscribe and group multicast services in many dimensions. They also developed tools for transforming standard Web services into ones that scales to clusters in data centers, automatically replicates data to improve peformance and reliability, and efficiently updates replicated data. Van Renesse and his group targeted the problem of hardening group communication systems to failures and attacks. They developed reliable techniques for detecting intruders in group communication systems and mitigating the damage that they can cause, such as providing incorrect information or dropping packets.

Gehrke and Demers investigated scalability and information management issues in the JBI and other military information dissemination systems. They worked on the challenges of building very large scale

databases which are spread among many sites and yet maintain strong forms of consistency and can be queried in a manner similar to the ways that centralized databases can be queried. They targeted emerging architectures where actual databases reside at the leaves, are updated using transactions, and can support triggered upcalls when events change in ways of interest. Within this overall effort, Gehke's group has been focused on extending existing database systems and building completely new ones optimized for use in distributed sensor networks and other systems that generate high volumes of data rapidly. Demers focused on understanding the behavior of systems built using probabilistic protocols and the risks associated with using approximate algorithms, developing a number of such algorithms for solving simple problems like solving distributed range queries over complex sensor networks, computing aggregate values in large systems, and optimal resource location in distributed settings.

Marzullo and Voelker in the UCSD group focused on providing high service availability and efficient reliability for large-scale distributed systems that form the foundation of JBI efforts. Voelker and his group addressed the problem of providing highly-available services in distributed systems composed of relatively unavailable components. A fundamental challenge for designing and building next-generation distributed systems is providing high availability using these highly unavailable components. Their work developed middleware that provides an abstraction of a highly-available platform to upper-layer software services while running on intermittently available components. They demonstrated this approach in a prototype wide-area file system. In terms of reliability, Marzullo and his group pursued the development of and insights into the theoretical understanding of distributed systems with dependent failures. Their work has developed new solutions to several well-known problems in distributed computing that are optimal when failures are not independent and do not have identical distributions. Making their theoretical results practical, they applied dependent failure system models to a system called Phoenix that cooperatively protects data from loss arising from Internet catastrophes from propagating malware such as viruses and worms.

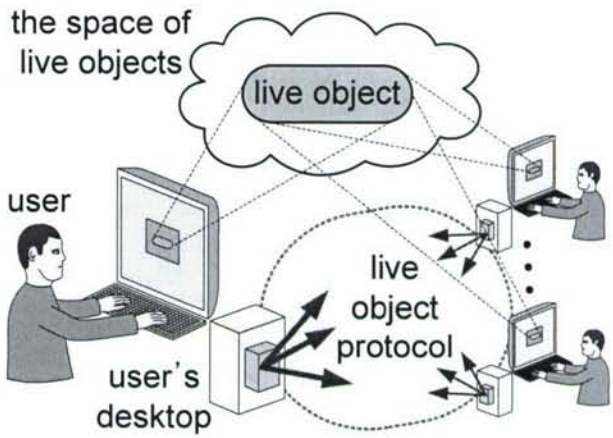
The JBI has proved to be an outstanding focus for our collaboration, and we embarked on a research agenda to use the MURI funding, together with other funding available to us directly from the JBI community and from other sources, such as DARPA, to assist AFRL in taking a major step forward on these challenging and extremely important questions. Over the period of the MURI grant, we established substantial groundwork for a scientific treatment of the military's pressing challenges, backed by rigorous experimental work that helps clarify the uncertainty concerning just how a JBI system might actually be implemented. We believe that the research results from this MURI effort will thus pave the way for future commercial efforts to provide concrete deliverables to the JBI and similar military projects in other services.

3. Status of Efforts, Accomplishments, and New Findings

To better encapsulate the major research accomplishments performed by our group, we next provide individual summaries of the goals, approaches, accomplishments, and impacts on military needs of our efforts funded through this MURI effort.

QUICKSILVER SCALABLE MULTICAST

Goal: *GIG/NCES platforms provide excellent support for point-to-point communication in a web services paradigm, but support for scalable group communication, data replication, distributed coordination and automated reaction to faults are sorely lacking. Our goal in the Quicksilver project is to overcome the inherent scalability problems that limited development of solutions having these properties. The basic premise is that if we can show how to scale a publish-subscribe or group multicast system in many dimensions, in a web services framework, we can then take the next step and build the missing tools.*



Approach: *We developed Quicksilver Scalable Multicast, and found a new way to embed the technology into the Windows and Linux platforms based on what we call a "live objects" interface. The key ideas were as follows:*

- *A live object extends the normal Windows support for component integration to permit a new kind of component in which members belong to a group that replicates data using high-speed multicast. Various back-end communication "drivers" can provide the multicast; QSM is just the first of these*
- *We constructed an implementation of such a multicast protocol, QSM, and have demonstrated that it can scale far beyond the limits of any prior multicast technology, particularly when large numbers of users employ the system and this results in a pattern of extensively overlapped multicast groups. QSM maintains extremely high performance even under disruptions and other forms of injected stress.*
- *We've begun to extend QSM with a new high-level language so that reliability can be layered over the core system in a simple, easily used manner that can support everything from best-effort reliability to strong models such as virtual synchrony or transactional one-copy serializability.*

Accomplishments: *Our work on QSM is attracting attention from major vendors such as Cisco and Microsoft, even as we develop a small user community of early adopters. With the expected release of our live objects layer in the fall of 2007, we should see a burst of users drawn to our system by the ability to create live documents and applications by dragging and dropping live objects onto web pages, into databases, or in Word documents. The basic idea is that by creating such a document and then sharing it, non-programmers can create sophisticated distributed applications much as they build PowerPoint presentations.*

We are also finding that the Web Services standards community is interested in our approach. A journal article proposing a way to extend WS-NOTIFICATION and WS-EVENTING to offer greater flexibility will appear in the fall.

Our work on Quicksilver will continue beyond the termination of the MURI effort under funding from AFRL, AFOSR and other sources.

Impact on the warfighter: Quicksilver enables a new kind of agile response to rapidly evolving conditions. Today, it would be impossible to imagine building, say, a customized application for a search-and-rescue mission on the fly, in the field. With live objects and Quicksilver it may be possible for tomorrow's commander to create custom information solutions as needed, in real-time, for instant deployment to the troops for whom accurate timely information can determine mission success.

References: (For downloads and complete list, visit http://www.cs.cornell.edu/projects/quicksilver/)

Scalable Publish-Subscribe in a Managed Framework. Krzysztof Ostrowski, Ken Birman. Cornell Technical Report (TR2007-2086). October, 2006.

The QuickSilver Properties Framework. Krzysztof Ostrowski, Ken Birman, Danny Dolev. OSDI'06 poster session, Seattle, WA, November 2006.

Properties Framework and Typed Endpoints for Scalable Group Communication. Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report TR2006-2062 (July, 2006).

Scalable Group Communication System for Scalable Trust. Krzysztof Ostrowski, Ken Birman. In Proceedings of The First ACM Workshop on Scalable Trusted Computing (ACM STC 2006). Fairfax, VA. November 3, 2006.

Extensible Web Services Architecture for Notification in Large-Scale Systems. Krzysztof Ostrowski and Ken Birman. In Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006). Chicago, IL, September 2006

Declarative Reliable Multi-Party Protocols Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Implementing High-Performance Multicast in a Managed Environment Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems. Ken Birman, Anne-Marie Kermarrec, Krzysztof Ostrowski, Marin Bertier, Danny Dolev, Robbert Van Renesse. Distributed Event Processing Systems and Architecture Workshop (DEPSA). June 2007.

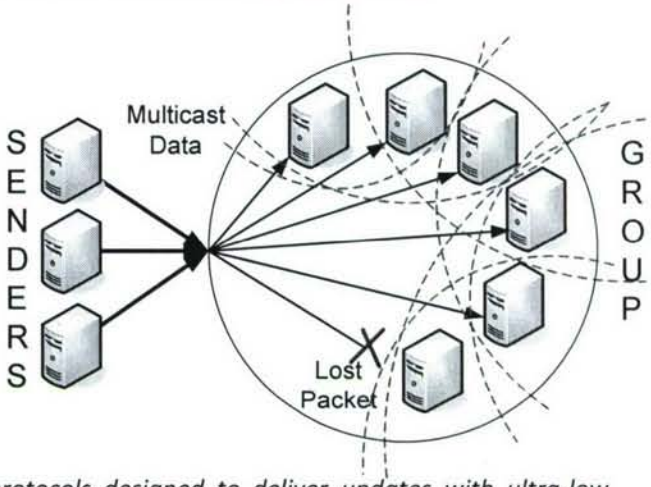
Live Distributed Objects: Enabling the Active Web Krzysztof Ostrowski, Ken Birman, Danny Dolev. To Appear in IEEE Internet Computing.

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. To Appear in the International Journal of Web Services Research. Volume 4, Number 4. October-December 2007.

RICOCHET SCALABLE TIME-CRITICAL MULTICAST PROTOCOL

Goal: *Data center systems are challenged by the difficulty of rapidly disseminating time-critical information, for example within a service running on multiple nodes in a cluster and where the data will trigger some real-time response. Such problems arise in many settings, including radar tracking systems, weapons targeting, real-time control of autonomous vehicles, etc. Moreover, there are many settings in which "standard" web services need to offer rapid end-user response time, even as updates flow into the core system.*



Approach: *Ricochet/Plato are a new family of multicast protocols designed to deliver updates with ultra-low latencies in clusters or data centers hosting scaled servers, again under the assumption that the services are implemented using web services standards. The key idea here is to aggregate traffic so that error correction codes can be computed more quickly than if each data stream was treated separately. Ricochet and Plato explore this in a multicast setting; we are currently taking the next step by developing, Maelstrom, which applies similar ideas with an emphasis on connections between data centers over long-distance WAN links. Cisco and Microsoft have shown keen interest in using these solutions in their respective platforms and products, and Raytheon is helping us explore transition into military platforms using the DDS standard.*

Accomplishments:

- *Designed and implemented the Ricochet protocol, undertook a comprehensive evaluation, and completed a series of papers on this work, including mobile wireless applications (Mistral).*
- *Through dialog with companies in the web services community, identified promising technology transition opportunities. Raytheon is taking the lead on pursuing these with us, focusing on the military-standard DDS communication architecture.*
- *Made our solutions available to other vendors, including the Apache web services platform.*
- *Developed and implemented a predictive real-time ordering protocol, Plato.*

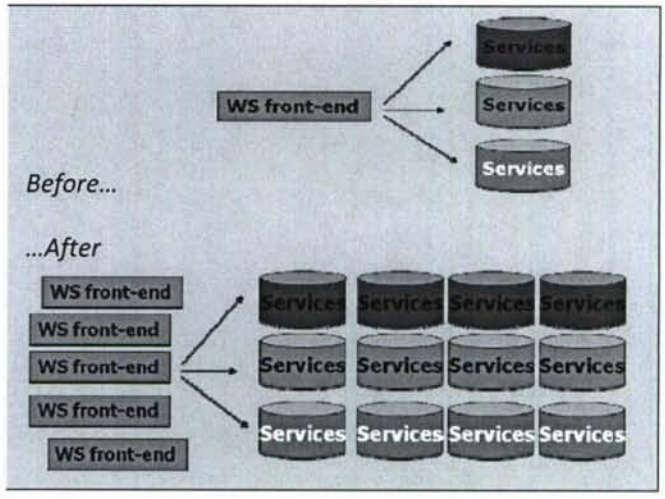
Impact on the warfighter: During military operations, timely information can have life or death implications. Ricochet makes possible dramatically improved responsiveness and maintains this guarantee as it scales up.

References: *Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*

Ricochet: Lateral Error Correction for Time-Critical Multicast. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. To Appear in Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). Cambridge, MA. April 2007.
Mistral: Efficient Flooding in Mobile Ad-hoc Networks. S. Pleisch, M. Balakrishnan, K. Birman, and R. van Renesse. In Proceedings of the Seventh ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM MobiHoc 2006). Florence, Italy May 2006.
PLATO:Predictive Latency-Aware Total Ordering. Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. In Proceedings of the SRDS 2006: 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK. October 2006.

TEMPEST: A TOOL FOR CREATING SCALABLE WEB SERVICES

Goal: *Today it is much too difficult for programmers with a typical MEng-level of training to implement scalable, self-managing web services that can run on datacenters in GIG/NCES settings and that will automatically adapt as conditions change. By solving this problem we can reduce the delays and costs associated with implementing new services for the GIG while also ensuring that those services will be seamlessly adaptive even under stress. Moreover, we can bring best-of-breed solutions to the table in a reusable form, reducing the tendency of vendors to produce stovepipe technologies that only the developer can extend or support.*



Approach: Tempest is a tool for turning a fairly vanilla web service into one that scales on a cluster or data center, has automatically replicated data, and employs Ricochet to send updates. Gossip communication is employed as an adjunct to this to repair any inconsistency that might arise as a result of a failure. Tempest is just reaching a stage at which early demos are feasible; the system is able to take a front end (for example an application that builds web pages) and a set of back end web services and will automatically replicate each of these, to varying degrees, in a manner that achieves predictable time-critical response even under stress, even when faults occur, and even when the services have very different behavioral profiles (such as mean response time, etc).

Accomplishments: Tempest was completed in early 2006 and works well; it uses a novel gossip-based approach to propagate updates, detect and repair inconsistencies, and for self-management of the clustered web application. In current work, we are exploring an extension in which Tempest could be used to create integrated enterprise web applications, still in a highly automated manner, with our gossip protocols used to extract data from source applications, replicate it across a WAN, and then integrate it with an application needing to import that data remotely.

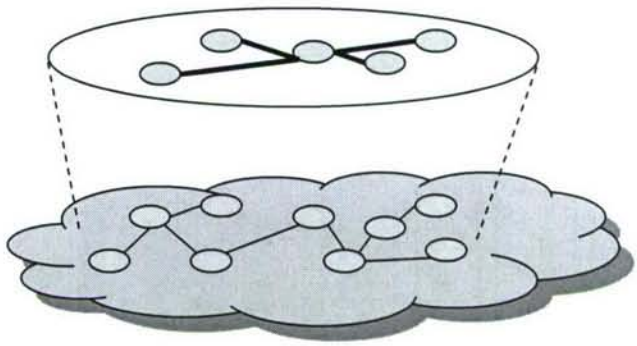
References: *Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

A Scalable Services Architecture. Tudor Marian, Ken Birman, and Robbert van Renesse. To appear in Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2006). Leeds, UK. October 2006.

Fireflies: Scalable Byzantine Overlay Networking

Goal: *"Fireflies" is a scalable protocol for supporting intrusion-tolerant network overlays. While such a protocol cannot distinguish Byzantine nodes from correct nodes in general, Fireflies provides correct nodes with a reasonably current view of which nodes are live, as well as a pseudo-random mesh for communication. The amount of data sent by correct nodes grows linearly with the aggregate rate of failures and recoveries, even if provoked by Byzantine nodes. The set of correct nodes form a connected sub-mesh, and Byzantine nodes cannot eclipse correct nodes.*



Approach: *Providing each member with membership is a form of agreement. Previous works on Byzantine fault-tolerant agreement establish invariants that are impossible to invalidate. Even the most practical of these protocols require several rounds of all members broadcasting state to all other members, and can consequently not scale up to more than perhaps a few dozen members. In order to scale to thousands or more members, we had to come up with a radically different approach. Fireflies makes use of epidemic techniques ("gossip") to form a probabilistic agreement, which can only establish invariants that hold with a certain probability. Because invariants never hold for certain, defense against adversaries trying to break agreement can never rest.*

Accomplishments: *Fireflies has been adopted by several research projects around the world. For example, at UT Austin Prof. Alvisi and Dahlin are working to create scalable Byzantine and Rational fault-tolerant communication systems making use of game theory (incentives). While their work has been successful, they were not able to deal this far with dynamic systems in which members could come and go. They are now building on Fireflies to remedy this shortcoming of their work. In Norway, Prof. Johansen has developed a system, based on Fireflies, to dispatch security updates in a timely and coordinated manner. Traditional software updates are vulnerable to reverse engineering to discover software flaws. In Israel, Prof. Dolev is developing a self-stabilizing version of Fireflies in order to add additional immunity to attacks. At Cornell itself, Fireflies forms the basis for the SecureStream video streaming system (reported separately).*

Impact on the warfighter: Modern warfighter equipment will almost certainly carry devices that employ state-of-the-art distributed communication protocols. If one or more such devices were compromised, most such protocols could be easily attacked without the warfighter being able to tell the difference. As a result, a warfighter cannot put much trust in these devices. Fireflies is much less susceptible to such attacks and as a result allows warfighters to put significantly more trust in devices that use Fireflies.

References: (For downloading Fireflies, visit http://sourceforge.net/projects/fireflies)

Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays. Håvard Johansen, Andre Allavena, and Robbert van Renesse. Eurosys 2006. Leuven, Belgium. April 2006.

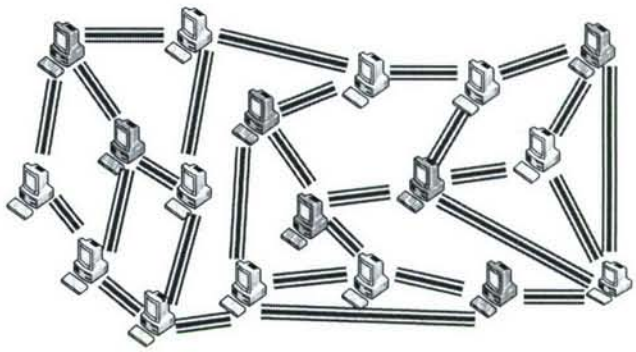
Correctness of Fireflies. Andre Allavena and Robbert van Renesse. Internal Report. June 2006.

FirePatch: Secure and Time-Critical Dissemination of Software Patches. Håvard Johansen, Dag Johansen, and Robbert van Renesse. IFIP International Information Security Conference (IFIPSEC 2007). Sandton, South-Africa. May 2007.

Self-Stabilizing and Byzantine-Tolerant Overlay Network. Danny Dolev and Robbert van Renesse. Submitted to OPODIS 2007. July 2007.

SecureStream: Intrusion-Tolerant Video Streaming

Goal: *Application-level multicast systems are vulnerable to attack that impede nodes from receiving desired data. Live streaming protocols are especially susceptible to packet loss induced by malicious behavior. We describe SecureStream, an application-level live streaming system built using a pull-based architecture that results in improved tolerance of malicious behavior. SecureStream is implemented as a layer running over Fireflies, an intrusion-tolerant membership protocol.*



Approach: *Our work introduces several techniques that reduce the opportunity for an attacker to compromise the quality of a streaming session, without incurring a high computational or network overhead. To repel forgery attacks, we employ an efficient packet authentication technique based on computing and distributing verification digests. To prevent attacks on the overlay structure (the membership protocol on top of which multicast systems operate), SecureStream is built upon Fireflies, a scalable one-hop Byzantine membership protocol.*

Accomplishments:

- *SecureStream is the first exploration of end-system attacks in the context of live streaming peer-to-peer protocols.*
- *We leverage previous work and present a comparison of different authentication protocols for signing and verifying packets efficiently in the context of application-level multicast.*
- *We thoroughly evaluate the effects of internal malicious peers on pull-based protocols. The issue is more serious than has previously been recognized.*
- *Aspects of our work are being adopted by other research groups. For example, at UT Austin Profs. Alvisi and Dahlin are using our linear digests. Prof. Porto at UFRGS in Brazil is extending SecureStream to deal with heterogeneity in the system.*

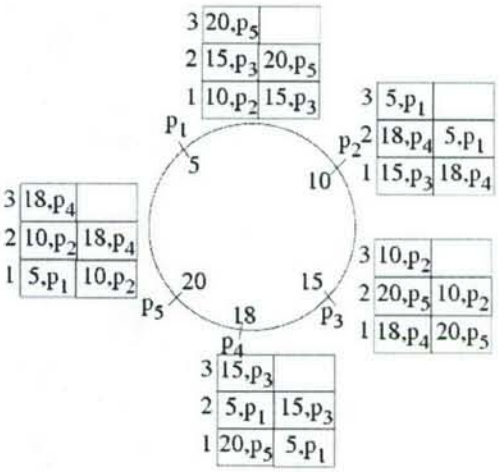
Impact on the warfighter: SecureStream can provide reliable delivery of streaming media to the warfighter even in the face of cyber-attacks.

References: (*Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*)

Maya Haridasan and Robbert van Renesse. *Defense Against Intrusion in a Live Streaming Multicast System*. 6th IEEE International Conference on Peer-to-Peer Computing (P2P2006). Cambridge, UK. September 2006.
Maya Haridasan and Robbert van Renesse. *SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination*. The Journal of Computer Communication's Special issue on Foundation of Peer-to-Peer Computing (accepted for publication). 2007.

P-RING: SCALABLE DISTRIBUTED RANGE QUERIES

Goal: *Modern data management systems are increasingly challenged by the need to support very large scale data sets distributed among many sites. The systems must allow distributed data to be queried in a manner similar to the ways centralized databases are queried. They must perform rapid dissemination of time-critical updates while providing strong consistency guarantees for concurrent queries, so that incoming data can reliably trigger a real-time response. Finally, the systems must be fault-tolerant, able to withstand high rates of churn with minimal effect on query performance or correctness.*



Approach:

We have developed P-Ring, a new peer-to-peer index structure that efficiently and scalably supports range queries as well as equality queries, and is robust even under high rates of churn. P-Ring can be viewed as an alternative approach to our earlier Kelips work. The Kelips design tolerates somewhat increased memory usage – O(sqrt(n)) – in exchange for O(1) file lookup times. This memory requirement is acceptable for current systems, but could eventually become a scalability bottleneck. P-Ring takes a different approach, using a hierarchy of fault-tolerant rings to provide O(log(n)) lookup time, by a protocol similar to a skiplist, with an improved memory requirement that is only polylog in n. The P-Ring also achieves excellent load balancing. A straightforward scheme yields a worst-case imbalance factor of 2, analogous to a B+ tree; while a more complicated but quite practical scheme can yield arbitrarily small imbalance factors with constant amortized overhead. A prototype implementation of P-Ring outperforms existing systems that attempt to provide similar guarantees.

Accomplishments:

Designed and implemented the P-Ring protocols and performed a thorough experimental evaluation.

- *Published the results in SIGMOD.*
- *Made our prototype implementation available as open source.*
- *Identified promising technology transition opportunities in collaboration with a local company (ATC-NY). Ongoing work is extending the system to a native XML database.*

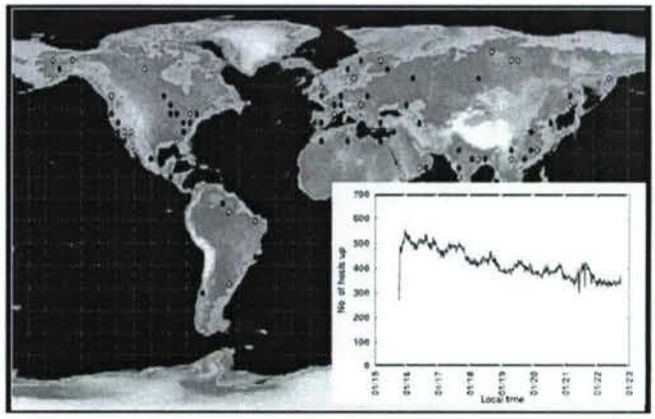
Impact on the warfighter: During military operations, timely and correct information can be vitally important. This work provides scalable correctness and responsiveness guarantees even under high rates of churn.

References:

P-Ring: An Efficient and Robust P2P Range Index Structure. Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke and Jayavel Shanmugasundaram. Proceedings 2007 ACM SIGMOD Conference. Beijing, China, June 2007.

Automated Availability Management

Goal: *A number of issues arise as Web Services and similar COTS components are migrated into GIG/NCES platforms. Our goal for developing automated availability management is to address the problem of providing highly-available services in distributed systems composed of relatively unavailable components. This problem arises in many settings, including ad-hoc wireless networks where disconnection and reconnection is frequent (e.g., battlefield scenarios), sensor networks (e.g., intelligence gathering), distributed computation and storage, etc.*



Approach: *A distinguishing feature across these disparate types of distributed systems is that they are composed of relatively unavailable components. Rather than being always available until failure, the components in these systems are both available and unavailable on a frequent basis (daily, even hourly), yet remain a functioning component of the system on long-term time scales (weeks to months to years). A fundamental challenge for designing and building next-generation distributed systems is providing high availability using these highly unavailable components. Automated availability management formalizes availability as an explicit property in distributed systems. Users and applications can request explicit availability guarantees for system objects and resources. For instance, in a wide-area distributed file system, users would specify that by default files require 99.9% availability over two years. To provide such guarantees, automated availability management uses (1) availability models to make efficient resource provisioning decisions and to predict and estimate future resource availability; (2) redundancy mechanisms to mask and tolerate component unavailability; and (3) repair policies to dynamically maintain resource availability in response to intermittent and permanent failure.*

Accomplishments: *We developed and evaluated a range of availability models, redundancy mechanisms, and repair policies across a spectrum of system configurations. As a concrete point in the system environment space, we measured the temporary and permanent failure characteristics of the Overnet peer-to-peer file sharing overlay network. This work was the first to study these characteristics in such systems, and the traces we gathered were used by many other research groups to evaluate their own efforts. We also refined automated availability management to exploit resource to improve system performance and reliability. Our goal has been to expose resource heterogeneity among nodes and tailor systems to take advantage of it.*

Impact on the warfighter: *Automated availability management provides a convenient abstraction for implementing highly available distributed services in situations where network disconnection and reconnection is frequent. The situations occur at many levels in military deployments, ranging from ad-hoc networks among troop patrols, battlefield communication networks linking troops, vehicles, and air support, carrier groups at sea, and the world-wide GIG IT infrastructure.*

References: (For downloads and complete list, visit http://sysnet.ucsd.edu/projects/recall/)

Replication Strategies for Highly Available Peer-to-Peer Storage Systems. Ranjita Bhagwan, David Moore, Stefan Savage, and Geoffrey M. Voelker. UCSD Technical Report No. CS2002-0726, November 2002.
Understanding Availability. Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Proceedings of the International Workshop on Peer To Peer Systems (IPTPS'03), Berkeley, CA, February, 2003.
Ranjita Bhagwan, Priya Mahadevan, George Varghese, and Geoffrey M. Voelker. Cone: A Distributed Heap-Based Approach to Resource Selection UCSD Technical Report CS2004-0784.
On Object Maintenance in Peer-to-Peer Systems Kiran Tati and Geoffrey M. Voelker. In Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS), Santa Barbara, California, February 2006.
Maximizing Data Locality in Distributed Systems, Fan Chung, Ron Graham, Ranjita Bhagwan, Geoffrey M. Voelker, and Stefan Savage, Journal of Computer and System Sciences 72(8), December 2006.

TotalRecall File System

Goal: *Our goal was to develop a specific system application of the automated availability approach. We designed a storage system called TotalRecall that applies automated availability management to large-scale, wide-area distributed storage systems. TotalRecall guarantees user-specified levels of data availability while minimizing the overhead needed to provide these guarantees in highly dynamic environments.*

Approach: *TotalRecall predicts the availability of its components over time, determines the appropriate level of redundancy to tolerate transient outages, and automatically initiates repair actions to meet user requirements. It uses replication and erasure coding to adapt the degree of redundancy and frequency of repair to the distribution of failures observed and predicted in the system. It also uses two repair strategies, eager repair and lazy repair, for trading off availability and replication overhead. Moreover, it closely approximates key system parameters, such as the appropriate level of redundancy, from a combination of underlying measurements and requirements. Finally, we have implemented and evaluated a prototype of TotalRecall that automatically adapts to changes in the underlying host population while effectively managing file availability and efficiently using bandwidth and storage resources.*

Accomplishments:

- *Developed and evaluated specific availability management mechanisms for storage systems, including availability models for short-term temporary and long-term permanent failures, erasure coding and mirroring redundancy mechanisms, and eager and lazy repair policies.*
- *Designed and implemented a prototype that runs on the PlanetLab testbed as the TotalRecall File System. Each participating host exports an NFSv3 file system interface to the system. External client hosts can mount TRFS and use it as any other remote NFS file system, storing data with high availability.*
- *Developed ``ShortCuts'', a routing approach for lookups that uses soft state to achieve routing performance that approaches the aggressive performance of one-hop schemes, yet uses an order of magnitude less communication overhead on average.*

Impact on the warfighter: *The TotalRecall File System provides a highly available distributed storage service in situations where network disconnection and reconnection is frequent. It is particularly useful in large-scale GIG IT infrastructures that face challenging communication constraints, such as among the many ships that comprise carrier groups operating at sea, as well as world-wide information infrastructure, such as the storage services supporting the data collection, analysis, and reporting performed by military branches and government agencies.*

References: (For downloads and complete list, visit http://sysnet.ucsd.edu/projects/recall/)

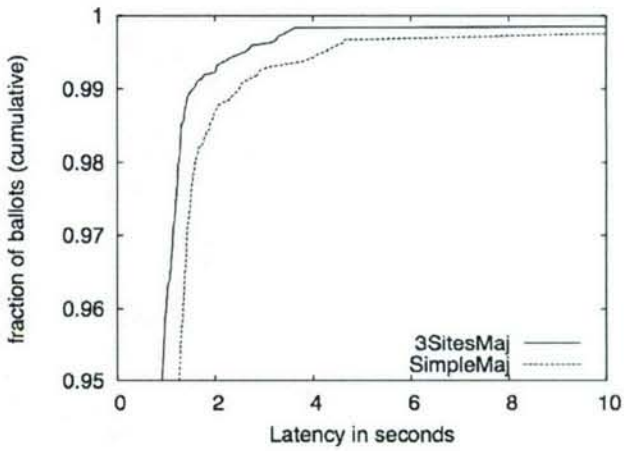
TotalRecall: System Support for Automated Availability Management Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. In Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, March 2004.

ShortCuts: Using Soft State To Improve DHT Routing. Kiran Tati and Geoffrey M. Voelker. In Proceedings of the Ninth International Workshop on Web Content Caching and Distribution (WCW'04), Beijing, China, October 2004.

Resource Reclamation in Distributed Hash Tables. Kiran Tati and Geoffrey M. Voelker. University of California, San Diego, CSE Technical Report CS2006-0863, July 2006.

Realistic Abstract Failure Models

Goal: *Failure models are part of the contract used in designing and deploying a distributed system. A failure model says what can go wrong with the environment, and so the system needs to be able to sustain its mission in the face of these adverse conditions. From a protocol design point of view, though, a failure model should be simple and abstract, since efficiency is obtained by leveraging off the details of the failure model. Practical details that are often ignored include non-identical failure rates, non-zero covariance of failures, and communications failures arising from BGP convergence issues.*



Approach: *One of the most fundamental protocols for fault tolerance is consensus, and so we deconstructed the various versions of this protocol to understand how it used the simple failure models for which it was developed. We identified two kinds of properties -* core *properties, useful describing failure scenarios, and* survivor set *properties, useful for showing that information is preserved despite failures. The two kinds of properties are duals of each other, and can be generalized to accommodate non-identical failures, non-zero covariance of failures, as well as many other failure patterns.*

Accomplishments: *We have generalized much of the lower bound results for consensus, quorum update, voting, and related problems. The results have been surprising: we have essentially developed a methodology for taking advantage of dependent failures. By knowing which failure patterns are more likely and which are not likely, replication of information can be done in an informed manner. In addition, we have found that some previous lower bounds are serendipitous: some difficulties with more general failure models appear only when dependent failures can occur. In addition, our work has shown that significant performance gains can be obtained using more accurate failure models. The graph above, for example, shows how a version of consensus has better availability and faster convergence time when the protocol makes use of the plausible failure patterns of a wide area network, even when no failures occur. Finally, we have developed a better understanding of the performance of core protocols in real*

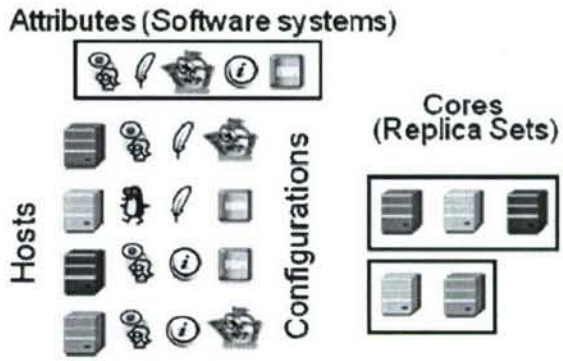
Impact on the warfighter: *Failures of an information system in a real military deployment are complex and not easily characterized using the simple failure models commonly used in high-level protocol design before our work. Using the simpler models results in solutions that are slower and require more infrastructure, both of which pose problems in a deployment. With our models and new versions of basic protocols, it should be possible to build more efficient and robust information system.*

References:

Synchronous Consensus for Dependent Process Failures, Flavio Junqueira and Keith Marzullo, Proceedings of the International Conference on Distributed Computing Systems (ICDCS), Providence, Rhode Island, May 2003.
The Virtue of Dependent Failures in Multi-site Systems, Flavio Junqueira and Keith Marzullo, Proceedings of the IEEE Workshop on Hot Topics in System Dependability (HotDep), Yokohama, Japan, June 2005.
Replication predicates for dependent-failure algorithms, Flavio Junqueira and Keith Marzullo, Proceedings of the 11th International Conference on Parallel and Distributed Computing (EuroPar), August 2005.
Coterie Availability in Sites, Flavio Junqueira and Keith Marzullo, Proceedings of the International Symposium on Distributed Computing (DISC), Cracow, Poland, September 2005.
Classic Paxos vs. Fast Paxos: *Caveat Emptor*, Flavio Junqueira, Yanhua Mao and Keith Marzullo. Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07), Edinburgh, UK, June 2007.

Informed Replication

Goal: *Informed replication is an application of our dependent failure models to real distributed systems. Large-scale distributed systems are highly vulnerable to Internet catastrophes: events in which an exceptionally successful network pathogen, like a worm or email virus, causes data loss on a significant percentage of the machines connected to the network. Informed replication takes advantage of software heterogeneity among nodes in a system to efficiently and reliably ensure that replicated data and services survive.*



Approach: *The key observation that makes informed replication both feasible and practical is that Internet epidemics exploit shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, a pathogen that exploits one or more vulnerabilities cannot cause all replicas to fail. For example, to prevent a distributed system from failing due to a pathogen that exploits vulnerabilities in Web servers, the system can place replicas on hosts running different Web server software.*

Accomplishments:

- *Developed a system model using our core abstraction to represent failure correlation in distributed systems. A core is a reliable minimal subset of components such that the probability of having all hosts in a core failing is negligible.*
- *Measured and characterized the diversity of the operating systems and network services of hosts in the UCSD network to evaluate the degree of software heterogeneity found in an Internet setting.*
- *Developed heuristics for computing cores that provide excellent reliability guarantees, have low overhead, bound the number of replica copies stored by any host, and the heuristics lend themselves to a fully distributed implementation for scalability.*
- *To demonstrate the feasibility and utility of our approach, we applied informed replication to the design and implementation of the Phoenix cooperative, distributed remote backup system.*

Impact on the warfighter: *Informed replication provides a powerful approach for enabling large-scale distributed systems to survive virulent, self-propagating Internet malware. Systems that include hosts with different software configurations can take advantage of the approach, including systems deployed in the battlefield as well as military and agency IT infrastructure.*

References:

The Phoenix Recovery System: Rebuilding from the Ashes of an Internet Catastrophe, Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker, Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, HI, May 2003, pages 73-78.

Surviving Internet Catastrophes, Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker, Proceedings of the USENIX Annual Technical Conference, Anaheim, CA, April 2005, pages 45-60.

4. Personnel Supported

Faculty, Researchers, and PostDocs

Ken Birman (Cornell), Alan Demers (Cornell), Richard Eaton (Cornell), Paul Francis (Cornell), Johannes Gehrke (Cornell), Christoph Koch (Cornell), Keith Marzullo (UCSD), Jayavel Shanmugasundaram (Cornell), Niki Trigoni (Cornell), Robbert Van Renesse (Cornell), Mark Riedwald (Cornell), Geoffrey M. Voelker (UCSD), and Werner Vogels (Cornell), Walker White (Cornell).

Graduate Students

Jeanne Albrecht (UCSD), David S. Anderson (UCSD), Ben Atkin (Cornell), Alvin AuYoung (UCSD), Anand Balachandran (UCSD), Mahesh Balakrishnan (Cornell), Hitesh Ballani (Cornell), Rimon Barr (Cornell), Leeann Bent (UCSD), Ranjita Bhagwan (UCSD), Adrian Bozdog (Cornell), Cristian Bucila (Cornell), John Calandrino (Cornell), Zhiyuan Chen (Cornell), Sigmund Cherem (Cornell), Jessica Chiang (UCSD), Adina Crainiceanu (Cornell), Annemarie Dahm (UCSD), Abhinandan Das (Cornell), Chris Fleizach (UCSD), Flavio Junqueira (UCSD), Lin Guo (Cornell), Indranil Gupta (Cornell), Maya Hardisasan (Cornell), Mingsheng Hong (Cornell), Ken Hopkinson (Cornell), Tibor Janosi (Cornell), Prakesh Linga (Cornell), Ashwin Machanavajjala (Cornell), Priya Mahadevan (UCSD), Yanhua Mao (UCSD), Marvin McNett (UCSD), Alper Mizrak (UCSD), Krzysztof Ostrowski (Cornell), Biswanath Panda (Cornell), Ishwar Ramani (UCSD), Venugopala Ramasubramanian (Cornell), Manpreet Singh (Cornell), Michael Sirivianos (UCSD), Kiran Tati (UCSD), Ruijie Wang (Cornell), Ming Woo-Kawaguchi (UCSD), Dmitrii Zagorodnov (UCSD), Xianan Zhang (UCSD), Xinyang Zhang (Cornell).

Undergraduate Students

Justin Koser (Cornell), Ben Kraft (Cornell), Amar Phanishayee (Cornell).

Interactions/Transitions

The members of the team have been very active, speaking at a wide range of conferences and workshops during the course of the MURI project. In particular, Professor Birman maintains active dialog and research collaborations with many companies, and also advises the US military and government in many capacities. During the period of the MURI effort he has had active dialog with the following companies and US government agencies. Obviously, some of these have been more substantive than others, but as a group, they illustrate the extent of ties between Birman's effort and industry and support his view that as technologies emerge suitable for potential technology transition, there will be good opportunities for pursuing them further. For example, at the current time, his group is working to explore a transition path for the Ricochet technology into Raytheon's DDS platform for military publish-subscribe applications that need real-time responsiveness.

- Air Force Office of the CIO, Air Force Research Laboratories, Apache Group, Amazon, Cisco, DARPA, DHS, Microsoft, Infosys, IBM, Intel, Google, Mitre, NSF, OSD / DDR&E, OSD / DDS&T, RAND, Raytheon, SRI, White House OSTP, WSO2

Honors and Awards

Ken Birman is a Fellow of the ACM.

Johannes Gehrke received an Alfred P. Sloan Fellowship (2003).

Geoffrey M. Voelker received the UCSD Hellman Faculty Fellowship (2002), the UCSD Chancellor's Associates Award for Excellence in Undergraduate Teaching (2006), and the UCSD Jacobs School Ericsson Distinguished Scholarship (2007).

5. Key Research Publications

Finally, we include key research publications from each of the projects summarized above to provide depth and detail of our MURI research efforts.

Extensible Web Services Architecture for Notification in Large-Scale Systems

Krzysztof Ostrowski
Cornell University
krzys@cs.cornell.edu

Ken Birman
Cornell University
ken@cs.cornell.edu

Abstract

Existing web services notification and eventing standards are useful in many applications, but they have serious limitations precluding large-scale deployments: it is impossible to use IP multicast or for recipients to forward messages to others and scalable notification trees must be setup manually. We propose[1] a design free of such limitations that could serve as a basis for extending or complementing these standards. The approach emerges from our prior work on QSM [1]*, a new web services eventing platform that can scale to extremely large environments.*

1. Introduction

1.1. Motivation

Notification is a valuable, widely used primitive for designing distributed systems. The growing popularity of RSS feeds and similar technologies shows that this is also true at Internet scales. The WS-Notification [2] and WS-Eventing [3] standards have been offered as a basis for interoperation of heterogeneous systems deployed across the Internet. Unlike RSS, they are subscription-based, and hence free of the scalability issues of polling, and support proxy nodes that can be used to build scalable notification trees. Nonetheless, they embody restrictions:

- *Not self-organizing*. While both standards permit the construction of notification trees, such trees must be manually configured and require the use of dedicated infrastructure nodes ("proxies"). Automated setup of dissemination trees, formed by recipients, and without the dedicated infrastructure, is more appropriate.
- *Inability to use external multicast frameworks*. Both standards leave it entirely to the recipients to prepare their communication endpoints for message delivery. This makes it impossible for a group of recipients to dynamically agree upon a shared IP multicast address, or to construct an overlay multicast within a segment of the network. Yet such techniques are central to achieving high performance and scalability, and could also be used to provide QoS guarantees or leverage the emergent technologies.
- *No forwarding among recipients*. Many content distribution schemes build overlays within which content recipients participate in message delivery. In current web services notification standards, however, recipients are *passive* (limited to data reception).
- *Difficult to manage*. At Internet scales, it is hard to create and maintain a dissemination structure that would permit any node to serve as a publisher or subscriber, for this requires many parties to maintain common infrastructure, agree on standards, topology and other factors. Any such large-scale infrastructure should respect local autonomy, whereby the owner of a portion of a network can set up policies for local routing, availability of IP multicast, etc.
- *Weak reliability*. Reliability in the existing schemes is limited to per-link guarantees, resulting from the use of TCP. In many situations, stronger guarantees are required, e.g. to support virtually synchronous, transactional or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages or participate in multi-party protocols, even weak guarantees cannot be provided.

1.2. Our Contribution

In this document, we propose a principled approach to building large-scale systems for web services notification. We outline a design for an extensible notification scheme free of the limitations just described, which is the basis for Quicksilver [1], a new scalable and reliable publish-subscribe and notification platform under development at Cornell. Motivated by the end-to-end principle, we separate the implementation of loss recovery and strong reliability properties from the unreliable dissemination of messages. Accordingly, our design includes a *reliability framework* and a *dissemination framework*: two largely independent, yet complementary structures.

Both frameworks reflect the principles articulated below, and they share many elements. In particular, both employ hierarchical protocol stacks, an idea that is central to our architecture. These stacks permit the definition of an Internet-scale loss recovery scheme which can employ different recovery policies within different administrative domains. Likewise, they permit a construction of a global dissemination scheme that uses different mechanisms to distribute data within different administrative domains.

[1] Our effort is supported by AFRL/Cornell Information Assurance Institute.

1.3. Design Principles

The limitations of the existing designs listed above and our experience designing scalable multicast systems led us to the following design principles:

- *Programmable nodes*. Senders and recipients should not be limited to sending or receiving. They should be able to perform certain basic operations on data streams, such as forwarding or annotating data with information to be used by other peers, in support of local *forwarding policies*. The latter must be expressive enough to support protocols used in today's content delivery networks, such as overlay trees, rings, mesh structures, gossip, link multiplexing, or delivery along redundant paths.
- *External control*. Forwarding policies used by nodes must be selected and updated in a consistent manner. A node cannot predict a-priori what policy to use, or which other nodes to peer with; it must permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering etc.
- *Hierarchical structure*. The principles listed above should apply to not just individual nodes, but also entire administrative domains such as LANs, data centers or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating the cooperation among organizations in maintaining the global infrastructure. The way a message is delivered to subscribers across the Internet thus reflects policies defined at various levels.
- *Isolation and local autonomy*. A certain degree of local autonomy of the administrative domains must be preserved; such as how messages are forwarded internally, which nodes create which communication endpoints etc. In essence, the structure of a domain should be hidden from other domains it is peering with and from the higher layers. Likewise, details of its own subcomponents should as opaque as possible.
- *Channel negotiation*. Communication channel creation should permit a handshake. A recipient might be asked to e.g. join an IP multicast group, or subscribe to an external system. The recipient could then make a configuration decision on the basis of the information about the sender, e.g. a LAN asked to prepare a communication endpoint for receiving may choose a well-provisioned node to handle the anticipated load.
- *Managed channels*. Communication channels should be represented as *active contracts* in which receivers have a degree of control over the way the senders are sending. In self-organizing systems, reconfiguration triggered by churn is common and communication channels often need to be reopened or updated to adapt to the changing topology, traffic patterns or capacities. For example, a channel that previously requested that a given source transmits messages to one node, might notify the source that messages should now be transmitted to two other nodes, instead.
- *Reusability*. It should be possible to specify a policy for message forwarding or loss recovery in a standard way and post it into an online library of such policies as a contribution to the community. Administrators willing to deploy a given policy within their administrative domain should be able to do so in a simple way, e.g. by drag-and-drop, within a suitable GUI.

1.4. Basic Concepts

We employ the usual terminology, where notifications are associated with *topics* and produced by *publishers* and delivered to *subscribers*. We use the term "*group X*" to refer to the group of nodes subscribed to topic "*X*". More than one publisher may exist for a given topic. The prospective publishers and subscribers register with a *subscription manager*, which can be decentralized and independent of the publishers (see Figure 1, top). In our architecture, nodes reside in *administrative domains*. Nodes in the same domain are jointly managed. It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries; either for administrative reasons, or because communication locally in a domain is cheaper than across domains, as it is often related to network topology. Publishers and subscribers may be scattered across organizations, which must cooperate in message delivery. This often presents a logistic challenge (see Figure 1, bottom).

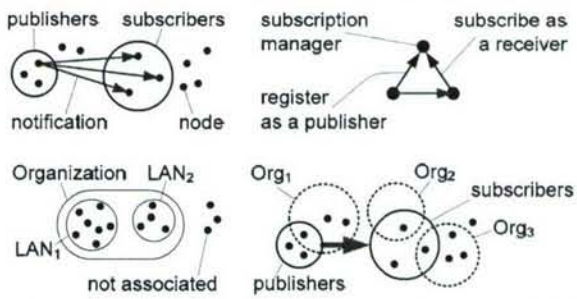


Figure 1. Publishers and subscribers register with the *subscription manager* (top). Nodes are scattered across *administrative domains* hierarchically divided into *sub-domains* (bottom).

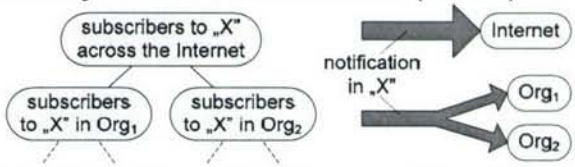


Figure 2. A hierarchical decomposition of the set of subscribers along the domain boundaries.

1.5. A Hierarchical View of the Network

A group X of subscribers for a given topic across the Internet can be divided into subsets Y_1, Y_2, ..., Y_N of subscribers in N top-level administrative domains (Figure 2). This can be continued recursively, leading to a hierarchical perspective on the set of all subscribers. By the principle of *isolation* and *local autonomy*, each administrative domain should manage the registration of its own publishers and subscribers internally, and decide how to distribute messages among them according to its local policy. Similar ideas were previously exploited in the context of content-based filtering [6], and in many scalable multicast algorithms, e.g. in RMTP [4]. This also reflects the principle of locality, implicit in many scalable protocols. Following this principle, groups of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection etc. Each such group is treated as a single cell within a global infrastructure. A protocol running at a global level connects all cells into a single structure. Scalability arises as in the *divide and conquer* principle. Additionally, the cells can locally share workload and amortize overhead, e.g. buffer messages coming from different sources and locally disseminate such combined bundles etc. We make heavy use of this property in our QSM [1] system.

This principle of locality and the hierarchical view of the network outlined above form the basis for our design.

2. Design Overview

2.1. The Hierarchy of Scopes

Our design is constructed upon the following principal concepts: *management scope*, *channel*, *filter*, *forwarding policy*, *session*, *recovery algorithm*, and *recovery domain*.

A *management scope (*or simply a *scope)* represents a *set of jointly managed nodes*. It may include a single node, span over a group of nodes residing within a certain administrative domain, or include nodes clustered based on other criteria, such as common interest. In the extreme, a scope may span the Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but it will often be the case, and we will refer to a *LAN scope* (or just a *LAN*) to mean the scope spanning over all nodes residing within a LAN. The reader might find it easier to understand our design with such examples in mind.

A scope is not just any group of nodes, the assumption that they are *jointly managed* is essential. The existence of a scope is dependent upon the existence of infrastructure that maintains its membership and administers it. For a scope that corresponds to a LAN, this could be a server managing all local nodes. In a domain that spans several data centers in an organization, it could be a management infrastructure with a server in the company headquarters indirectly managing the network via subordinate servers in data centers. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the global scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many independently managed global scopes could thus co-exist.

Like administrative domains, scopes form a hierarchy, defined by a relation of *membership*: a scope may *declare* itself to be a *member* (*sub-scope*) of another scope. If X declares itself to be a member of Y, it means X is either physically or logically a part (or subset) of Y. Typically a scope defined for a sub-domain X of some administrative domain Y will be a member of the scope defined for Y. For instance, a node would be a member of a LAN. The LAN would be a member of a data center, which in turn would be a member of a corporate network etc. A node could also be a member of a scope of some overlay network. For a data center, two scopes might be defined, e.g. a monitoring scope and a control scope, both scopes covering the entire data center, with some LANs being a part of one scope, the other scope, or both. The corporate scope could be a member of several Internet-wide scopes.

The generality in these definitions allows us to model various special cases, such as clustering of nodes based on interest or other factors. Such clusters, formed e.g. by a server managing a LAN and based on node subscription patterns, could also be considered scopes, all managed by the same server. Nodes would be members of clusters and clusters (not nodes) would be members of the LAN. As it will be explained below, each cluster, as a separate scope, could be locally and independently managed. In [1], such construction is a basis for our scalable multicast protocol.

A scope hierarchy is not a tree. There may be multiple global scopes, or many super-scopes for any given scope. However, a scope always decomposes into a tree of subscopes, down to the level of nodes. We refer to a *span of a scope X* as the set of all nodes at the bottom of a hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, i.e. such that all subscribers to X reside in the span of X. Publishing a message in a topic is thus equivalent to delivering it to all subscribers in the span of some global scope, which may be further decomposed into subscribers in the spans of sub-scopes (compare section 1.5 and Figure 2).

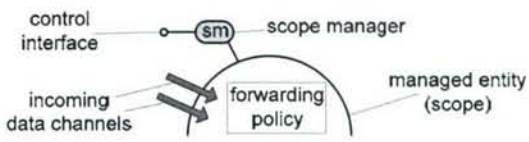


Figure 3. Accessed via a control interface and configured with a forwarding policy, a scope manager creates incoming data channels.

2.2. The Anatomy of a Scope

The infrastructure managing a scope is referred to as a *scope manager* (SM). A single SM may control multiple scopes. It may be hosted on a single node, or on a set of nodes, perhaps outside of the scope it controls. It exposes a *control interface*, a web service hosted at a well-known address, to dispatch control requests directed to scopes it controls (Figure 3). SMs interact by calling each other's control web interfaces (see also [8]).

A scope maintains communication *channels* for use by other scopes. A *channel* is a mechanism through which a message can be delivered to all those nodes in the span of this scope that subscribed to any of a certain set of topics. In a scope spanning over a single node, a channel may be just an address/protocol pair; creating it would mean arranging for a local process to open a socket. In a distributed scope, a channel could be an IP multicast address; creating it would require all local nodes to listen at this address. In an overlay network, a channel could lead to nodes that forward messages across the entire overlay.

A scope that spans over a set of nodes is governed by forwarding *policy* specifying how messages that originate within that scope or arrive through some communication channel are forwarded internally and to other scopes.

The reader will recognize in our construction the principles we formulated earlier. Scopes, whether individual nodes, LANs or overlays, are *externally controlled* using their control interfaces, may be *programmed* with policies that govern the way messages are distributed internally and forwarded to other scopes, and transmit messages via *managed* communication channels established through a dialogue between a pair of SMs.

2.3. Hierarchical Composition of Policies

Following our design principles, we propose to solve the issue of a large-scale global cooperation in message delivery between independently managed administrative domains by introducing a hierarchical structure, in which forwarding policies defined at various levels are merged into a single dissemination scheme. Each scope is configured with a policy dictating, on a per-topic (and perhaps a per-sender) basis, how messages are forwarded among its members. For example, a policy governing a global scope might determine how messages in topic T, originating in a corporate network X, are forwarded between the various organizations. A policy of a scope of the given organization's network might determine how to forward messages among its data centers, and so on. A policy defined for a particular scope X is always defined at the granularity of X's members (not individual nodes). The way a given sub-scope Y of X delivers messages internally is a decision made by Y autonomously. Similarly, X's policy may specify that Y should forward messages to Z, but it is up to Y's policy to determine how to perform this task.

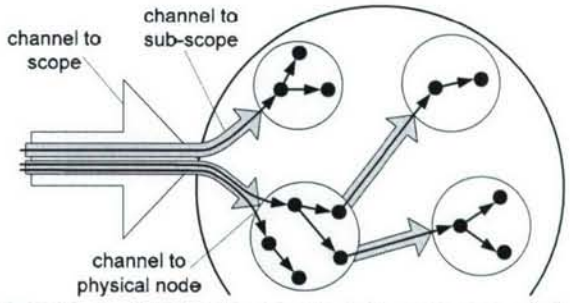


Figure 4. Channels created in support of forwarding policies defined at different levels.

Figure 5. Forwarding graphs for different topics are superimposed. Two members may be linked by multiple channels, each with a different filter.

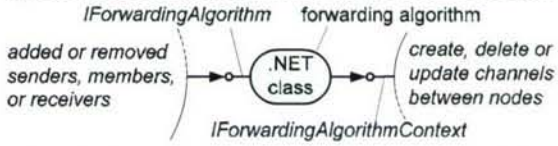


Figure 6. A forwarding policy as a code snippet.

Accordingly, a global policy may request organization X to forward messages in topic T to organizations Y and Z. A policy governing X may then determine that to distribute messages in X, they must be sent to LAN_1, which will forward them to LAN_2. The same policy might also specify which LANs within X should forward to Y and Z. Finally, the policies of the respective LANs could delegate these tasks to individual nodes. When the policies defined at all the scopes involved are combined, the resulting *forwarding structure* completely determines the way messages are forwarded (see Figure 4).

In the examples above, the policies are simply graphs of connections: each message is always forwarded along every channel. In general, however, each channel may be optionally constrained by a *filter* that decides, on a per-message basis, whether to forward or not, and optionally tags the message with custom attributes. This allows us to express many popular techniques, e.g. using redundant paths, multiplexing between dissemination trees etc.

Every scope manager maintains internally a mapping from topics to policies: a graph of channels to create and filters on them. Such graphs of connections for different topics are superimposed (see Figure 5). Based on this, the SM asks the scope members to create channels and filters. When the structure is modified as a result of membership or subscription changes, the SM makes additional control requests to reflect this.

In our framework, a policy is defined as an algorithm that lives in an *abstract context*, with a fixed set of *events*

to respond to, standard set of *operations* and *attributes* to inspect. In a prototype we are developing, we implement a forwarding policy as a .NET class, stored in a DLL on an *algorithm library* server, that implements an abstract interface and interacts with an abstract *context* hiding the details of the environment (Figure 6). This allows our policies to be used within any scope.

2.4. Communication Channels

Consider a node X, a member of a scope Z that, based on a forwarding policy at Z, has been requested to establish a communication channel to scope Y to forward messages in topic T. Following the protocol, X asks the SM of Y for the specification of the channel to Y that should be used for messages in topic T. The SM of Y might respond with an address/protocol pair that X should use to send over this channel. Alternatively, a forwarding policy defined for T at scope Y may dictate that, in order to send to Y in topic T, X should establish channels to members A and B of Y, constrained with filters α and β. After X learns this from the SM of Y, it contacts SMs of A and B for details. Notice how the channel decomposes into sub-channels to A and B through a policy at a target scope Y.

This decomposition continues hierarchically, until the point when scope X is left with a tree containing filters in internal nodes and address/protocol pairs at the leaves (Figure 9). In order to send a message along the channel built in this way, X executes filters to determine which sub-channels to use, proceeding recursively, until it is left with a list of address/protocol pairs, then transmits the message. Filters will typically be simple, such as modulo-**n**; hence X could perform this procedure very efficiently.

Accordingly, to support the hierarchical composition of policies described in the preceding section, we define a channel as one of the following: an address/protocol pair, a reference to an external multicast mechanism, or a set of sub-channels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which sub-channels to use for sending, on a per-message basis (see Figure 7 and Figure 8).

Consider now the case when scope X, spanning over a set of nodes, has been requested to create a channel to scope Y. Through a dialogue with Y and its sub-scopes, X can get a detailed channel definition, but unlike in the example above, X now spans over a set of nodes, and as such, it cannot *execute* filters or *send* messages.

We propose two example generic techniques that solve this problem: *delegation* and *replication* (Figure 10). Both rely on the fact that if scope X receives messages in a topic T, then some of its members, Z, must receive them (for otherwise X would not be made part of a forwarding structure for topic T by X's super-scope). In case of *delegation*, X requests such a sub-scope Z to create the channel on behalf of X, essentially delegating the whole channel. The problem can be recursively delegated, down to the level where a single physical node is requested to create a channel. A more sophisticated use of delegation would be for X to delegate sub-channels. In such case, X would first contact Y to obtain the list of sub-channels and the corresponding filters, and for each of these sub-channels, delegate it to one of its sub-scopes. In any case, X delegates the responsibility for sending over a channel, in one way or another, to one or more of its sub-scopes.

In case of *replication*, scope X requests **n** of its sub-scopes to create the channel, but constrains each with a modulo-**n** filter based on a message sequence number (i.e. sub-scope **k** only forwards messages with numbers **m** such that **m** mod **n** equals **k**), effectively implementing a round-robin policy. Although all sub-scopes would create the same channel, the round-robin filtering policy ensures that every message is forwarded only by one of them.

Figure 7. A channel may be an *address/protocol* pair (left), or it may consist of *sub-channels*, with an *algorithm* deciding what goes where (right).

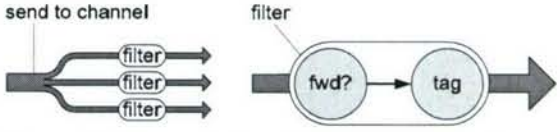


Figure 8. Channel algorithms are realized as sets of filters, one per subchannel, deciding whether to forward, and optionally adding custom tags.

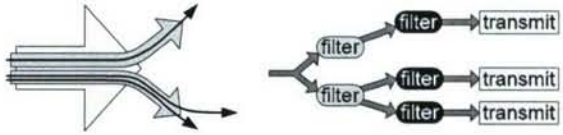


Figure 9. A channel split into sub-channels and a possible filter tree corresponding to it.

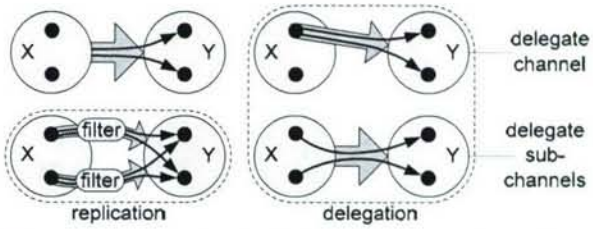


Figure 10. A distributed scope may delegate a channel or its sub-channels to members, or it may replicate them among members with filters that jointly implement a round-robin policy.

2.5. Reliability Scopes

The design of the reliability framework also relies on the concept of management scopes, referred to here as *reliability scopes* (in contrast to *dissemination scopes* in the dissemination framework). A reliability scope isolates and encapsulates the local aspects of loss recovery, hiding details from other scopes, just like a dissemination scope hides the local aspects of message delivery. Reliability scopes are also controlled by scope managers. Both kinds of scopes would typically overlap. For example, a single scope could be defined for an administrative domain such as a LAN, isolating local aspect of both dissemination and reliability. The scope could then be controlled by a single SM managing both dissemination and reliability.

The separation of dissemination from reliability makes it possible to combine an arbitrary unreliable notification mechanism, such as IP multicast or an overlay content delivery system, with a wide range of reliability protocols expressible in our reliability framework. This degree of reusability has not been possible with prior architectures.

2.6. Hierarchical Approach to Reliability

Our approach to reliability resembles our approach to dissemination. Just as channels are decomposed into sub-channels, in the reliability framework we decompose the task of repairing after message losses and providing other reliability goals. Recovering messages in a certain scope is modeled as recovering within sub-scopes, and then recovering "among" the sub-scopes (Figure 11). Just like recovery among single nodes, recovery among LANs may involve comparing their "state" (such as aggregated ACK or NAK information for the entire LANs) and forwarding lost messages. In section 2.9 we give examples of how recovery protocols may be defined and combined.

In our framework, different recovery schemes may be used in different scopes, to reflect differences in network topology, node or network capacity, the way subscribers are distributed (e.g. clustered vs. scattered around) etc.

Just like messages are disseminated through channels, reliability is achieved via *recovery domains*. A recovery domain **D** in scope **X** may be thought of as a "distributed recovery protocol running among some nodes in **X**" that performs recovery-related tasks for a certain set of topics. The concept of a recovery domain is symmetric, dual to the notion of a channel. We present it via analogy.

- Just like a channel is created to disseminate messages for some topics $\mathbf{T_1}$, $\mathbf{T_2}$, ... $\mathbf{T_k}$ in scope **X**, a recovery domain is created to handle loss recovery and other reliability tasks, again for a specific set of topics and in a specific scope. Just like there may be multiple channels to a scope, e.g. for different sets of topics, multiple recovery domains, each for different topics, may exist within a single reliability scope.
- Just like channels may be composed of sub-channels, a recovery domain **D** defined at scope **X** may be composed of *sub-domains* $\mathbf{D_1}$, $\mathbf{D_2}$, ... $\mathbf{D_n}$ defined at sub-scopes of **X** (we will call them *members* of D). Each such sub-domain $\mathbf{D_i}$ handles recovery for a certain set of subscribers in the respective sub-scope, while **D** handles recovery "across" its sub-domains.
- Just like channels are composed of sub-channels via applying filters assigned by forwarding policies, a recovery domain **D** performs its recovery tasks using a *recovery algorithm*. Such an algorithm, assigned to **D**, specifies how to combine recovery mechanisms in the sub-domains of **D** into a mechanism for all of **D**. Recovery algorithms are defined in terms of how the sub-domains "interact" with each other. We will see how this is achieved in section 2.9.
- Just like a single channel may be used to disseminate messages in multiple topics, a recovery domain may run a single protocol to perform recovery for multiple topics. In both cases, reusing a single mechanism (a channel, a token ring etc.) may significantly improve performance due to the reduction in the total number of "control" messages. We evaluated this idea in [1].

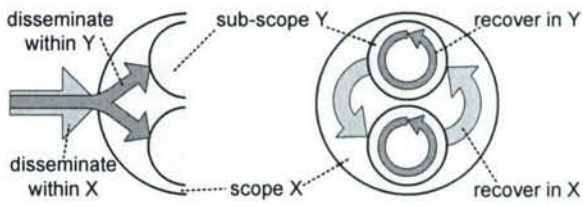


Figure 11. The similarity between hierarchical dissemination (left) and recovery (right).

Each individual node is a recovery domain on its own. In a distributed scope such as a LAN, on the other hand, two cases are possible. First, a single domain may cover the entire LAN. All internal nodes could form e.g. a token ring, exchange ACKs for messages in all topics, and use this to arrange for local repairs. Another possibility is that separate domains would be created for every individual topic. Subscribers to different topics would form separate structures, such as ring or trees, and run separate protocol instances in each, exchanging state and los messages.

As explained later, recovery domains in our system actually handle recovery for specific *sessions*, not just for specific topics. Sessions are introduced in section 2.7.

A recovery domain D of a data center could have as its members recovery domains created in LANs. Note that in this case, members of D would be sets of nodes. A recovery algorithm running in D would specify how all these different sets of nodes should exchange state and forward lost messages to one another. Note the similarity to a forwarding policy in a data center, which would also specify

how messages are forwarded among sets of nodes. As shown in section 2.10, recovery algorithms are implemented through delegation, just like forwarding. A concept of a *recovery algorithm* is, to a certain extent, symmetric to the notion of a forwarding policy.

2.7. Sessions

Within our architecture, protocols that provide strong reliability guarantees express them in terms of *epochs*. An epoch corresponds to what in group communication literature is called a *membership view*. The lifetime of a topic is divided into a sequence of epochs. Whenever the set of subscribers to a topic changes as a result of a subscribe/unsubscribe request or a failure, the event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs. One then defines consistency in terms of which messages may be delivered to which subscribers and at what time, relative to epoch boundaries. The traditional term "membership view" reflects the fact that epochs begin and end with membership change events. The set of subscribers during a given epoch is fixed.

Although simple protocols, such as SRM or RMTP, do not rely on a consistent view of group membership, and their properties are not defined in terms of epochs, epochs are still a useful, if not a universal concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, e.g. to rebuild a distributed structure, release resources or cancel activity that is no longer necessary. Many simple protocols simply do not take this factor into account.

We introduce the idea of a *session*, a generalization of an epoch (membership view). A session is also an epoch in the prior sense, i.e. the lifetime of any given topic can always be divided into a sequence of sessions. Like before, any membership change marks the beginning of a new session and for a given session, membership is fixed. However, a new session may also be initiated even if membership is unchanged. The reliability properties of a group may vary to some extent in the subsequent sessions. An important example is an administrative change, where a new protocol is introduced, e.g. because it is more efficient or to fix a bug in the existing protocol. In Internet-scale systems such administrative changes must be performed online; session changes achieve this.

Session numbers are assigned globally for consistency. As explained before, for a given topic, a single "global" scope always exists such that all subscribers to that topic reside within the span of this scope. This is true for both dissemination and reliability frameworks. Usually, both global scopes overlap and are managed by a single SM. The top-level SM assigns and updates session numbers. Note that local topics (e.g. internal to an organization) could be managed by the local SM, much in a way local newsgroups are visible and managed locally.

Before discussing the mechanisms used to manage membership, we conclude the discussion of sessions by explaining how they impact the behavior of publishers and subscribers. After registering, a publisher waits for the SM to notify it of the session number to use for a particular topic. A publisher is also notified of changes to the session number for topics it registered with. All published messages are tagged with the most recent session number, so that whenever a new session is started for a topic, within a short period of time no further messages will be sent in the previous session. Old sessions eventually quiesce as receivers deliver messages and the system completes flushing, cleanup and other reliability mechanisms used by the particular protocol. Similarly, after subscribing to a topic, a node does not process messages tagged as committed to session **k** until it is explicitly notified that it should receive messages in that session. Later, after session **k+1** starts, all subscribers are notified that session **k** is entering a *flushing* phase (this term originates in *virtual synchrony* protocols, but similar mechanisms are common in reliable protocols; a protocol lacking a flush mechanism simply ignores such notifications). Eventually, subscribers report that they have completed flushing and a global decision is made to cease activity and *cleanup* resources pertaining to session **k**, completing the transition.

2.8. Constructing the Recovery Structure

Reliable protocols often rely on, or could benefit from, a consistent view of membership. It helps to determine which nodes have crashed or disconnected. In existing systems, this is achieved by a Global Membership Service (GMS) that monitors failures and membership changes, decides when to install new membership views for topics, and notifies all affected members of the new views. In our framework, the global SM for a given topic is responsible for announcing when sessions begin and end. However, if the global SM had to process all subscriptions, it would lead to a non-scalable design that violates the principle of isolation. To avoid this, for each topic **T** we distribute the information about membership of **T** across all SMs in the hierarchy of scopes for **T** (this hierarchy was defined in section 2.1). Each SM thus has only a partial membership view for each session. This scheme is outlined below.

In the reliability framework, if a scope **X** subscribes to a topic **T**, it specifies some local recovery domain **D** that should handle the recovery for topic **T** in **X**. The **X**'s super-scope **Y** processes this subscription request jointly with requests from other sub-scopes. It then creates its own recovery domains, with the newly subscribed and perhaps some existing sub-domains as members, and then issues its own subscription requests to its super-scope. This continues recursively up to the global scope.

The scheme used by the super-scope to create recovery domains must abide by three rules. First, the list of sub-

domains of a recovery domain is determined once at the time of creation, and fixed throughout its lifetime. This is necessary to ensure that a hierarchical structure employed for recovery in any given session does not change, which simplifies the overall design. Second, a recovery domain **D** at scope **X** is responsible for handling recovery for a specific set of topics, in specific sessions. If a change in membership in any of these topics occurs locally in **X**, a new recovery domain **D'** must be created, and when a new session is announced, it is installed in **D'**. This is because the existing recovery domain **D** no longer represents the current set of subscribers within **X**, hence a new distributed structure **D'** must be established. Finally, if a new session is announced for some topic **T**, but no membership changes occurred for **T** within scope **X** since the previous session, then an existing recovery domain should be re-used to handle recovery in the new session.

In a scope in which recovery for each topic is handled individually, we would maintain a separate sequence of recovery domains for each topic. A new domain would be created whenever the set of subscribers locally changes. In a scope in which recovery for all topic is performed jointly, such as e.g. in a cluster of nodes defined based on subscription patterns in which all nodes are subscribers to the same set of topics, there will be just a single sequence of recovery domains. We used the latter scheme in [1].

The above procedure effectively constructs a hierarchy of sub-domains, with the property that for each topic **T**, the recovery domains subscribed to **T** form a tree.

The global scope assigns new session numbers for all topics for which subscribe or unsubscribe requests have been received, and determines which of its local recovery domains should handle the new sessions. This represents a coarse-grained membership view, for each session only top-level recovery domains are specified, with no further details. The information about the new sessions is now sent down the tree of subscribers, and transformed along the way to filter out unnecessary details. The membership information a scope **X** receives for a session **S** is limited to one level "above" **X**, i.e. it includes **X**'s own recovery domain that got subscribed to **S** and the recovery domains of its *sibling* scopes (i.e. scopes that have the same super-scope). It is also coarse-grained, i.e. it does not provide any details at the level "below" **X** or its siblings.

2.9. Modeling Recovery Algorithms

The design of the reliability framework is based on an abstract model of a distributed protocol dealing with loss recovery and other reliability properties. When expressed within our framework, such protocols will be referred to as *recovery algorithms*. Recovery algorithms are the basic building blocks in constructing our hierarchical reliability protocols, much in a way channels and filters are the basic building blocks in our forwarding infrastructure.

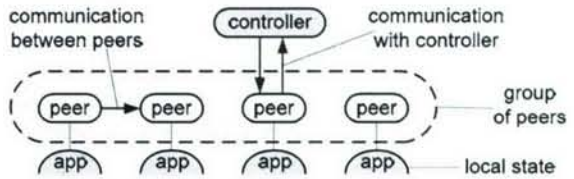


Figure 12. A group of *peers* in a reliable protocol.

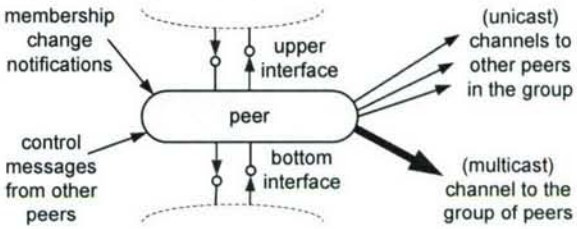


Figure 13. A peer modeled as a component living in abstract environment (events, interfaces etc.).

A protocol such as SRM, RMTP, or virtual synchrony is defined in terms of a group of cooperating *peers* that send control messages and forward lost packets to each other, and perhaps to a distinguished node, such as a sender or some node higher in a hierarchy, that we will refer to as a *controller* (Figure 12). The controller does not have to be a separate node; this function could be served by one of the peers. The distinction between the peers and the controller may be purely functional. The point is that the group of peers, as a whole, may be asked to perform a certain action, or calculate a value, for some higher-level entity, e.g. a sender, a higher-level protocol, or a layer in a hierarchical structure etc. Examples of such actions include requesting or performing a retransmission for all nodes, reporting which messages were successfully delivered to all nodes etc. Irrespectively of how exactly the interaction with a controller is realized, it is present in this form or another in almost every protocol run by a set of receivers. We shall refer to it as an *upper interface*.

Each peer inspects and controls *local state*. Such state may include e.g. a list of messages received and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered etc. Operations a peer may issue to change the local state could include e.g. retrieving/purging messages from a local cache, marking messages as "deliverable", handing a previously missed message to the application or assigning message sequence in a "totally ordered" group. We refer to such operations, used to view or control local state, as a *bottom interface*.

In protocols offering strong guarantees, peers typically know the membership of their group, received as a part of the initialization process, and subsequently updated via *membership change* events. Peers send control messages to each other to share state or to request actions, such as forwarding messages. Sometimes, as in SRM, a multicast channel to the entire peer group exists.

To summarize, in most reliable protocols a peer can be modeled as running in an environment that provides the following: a *membership view* of its peer group, *channels* to all other peers, and sometimes to the entire group, a *bottom interface* to inspect or control local state, and an *upper interface* to interact with a sender or a higher level in the hierarchy concerning the state of the whole group, (Figure 13). In some protocols, parts of the environment might be unavailable, e.g. in SRM peers might not know other peers. The bottom and upper interfaces would vary.

This model is flexible enough to capture the key ideas and features of a wide class of protocols, including virtual synchrony. However, because in our framework protocols must be reusable in different scopes, they may need to be expressed in a slightly different way, as explained below.

In the RMTP protocol [4], the sender and the receivers for a given topic form a tree. Within this tree, each subset of nodes consisting of a parent and child nodes serves as a separate, local recovery group. The child nodes in every such group send their local ACK/NAK information to the parent node, which arranges for a local recovery within the recovery group. The parent itself is either a child node in another recovery group, or it is a sender, at the root of the tree. Packet losses in this scheme are recovered on a hop-by-hop basis, top-down or bottom-up, one level at a time. This scheme distributes the burden of processing the individual ACKs/NAKs, and of retransmissions, which is normally the responsibility of the sender. This improves scalability and prevents the "ACK implosion".

There are two ways to express RMTP in our model. One approach is to view each recovery group consisting of a parent node and its child nodes as a separate group of peers (Figure 14). Since internal nodes in the RMTP tree simultaneously play two roles, a "parent" node in one recovery group and a "child" node in another, we think of a node as running two "agents", each representing a different "half" of the node and serving as a peer in a separate peer group. Every group of peers, in this perspective, includes the "bottom agent" of a parent node and "upper agents" of child nodes. When a node sends messages to its child nodes as a result of receiving a message from its parent, of vice versa, we may think of those two "agents" as interacting with each other through a certain interface that one of them views as upper, and the other as bottom. These two agents play different roles, as explained below.

The bottom agent of each node interacts via its *bottom interface* with the local state of the node. It also serves as a distinguished peer in the peer group composed of itself and the upper agents of child nodes. A protocol running in this peer group is used to exchange ACKs between child nodes and the parent node and arrange for message forwarding between peers, but also to calculate collective ACKs for the peer group, i.e. which messages were not recoverable in the group. This is communicated by the bottom agent, via its *upper interface*, to the upper agent.

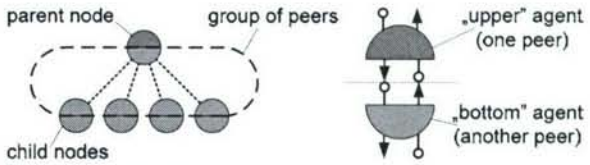


Figure 14. RMTP expressed in our model. A node hosts "agents" playing different roles.

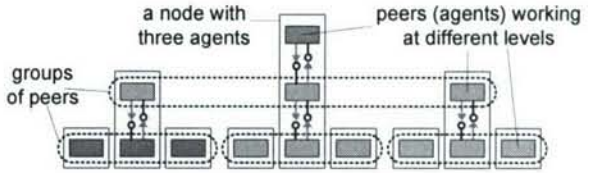


Figure 15. Another way to express RMTP. Each node hosts multiple "agents" that act as peers at different levels of the RMTP hierarchy.

The upper agent of every node interacts via its *bottom interface* with the bottom agent. What the upper agent considers as its "local state" is not the local state of the node. Instead, it is the state of the entire recovery group, including the parent and child nodes, that is collected for the upper agent by the bottom agent.

Such interactions, between a component that is a part of a "higher layer" and a component that resides in a "lower layer", both components co-located on the same physical node and connected via their upper and bottom interfaces, are the key element in our architecture.

At the top of this hierarchy, the upper agent of the root node communicates through its *upper interface* the state of the entire tree of receivers to the sender.

The second way to model RMTP captures the essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node may host multiple agents, connected to each other, each working at a different level (Figure 15). In a LAN scope, all nodes host a "local agent" component (green), similar to the "bottom agents" above, that serves as a peer in the group of all LAN nodes. The *bottom interface* used by this agent interacts with the local state. These peers exchange ACKs and arrange for message forwarding, with one of them acting as a "parent" and all other as "children". On the node hosting the "parent", a "higher-level" agent is hosted (orange); we refer to it as a "LAN agent", for there is exactly one in each LAN, and it represents the entire LAN. It connects through its bottom interface to the local agent, which is a distinguished peer in a LAN peer group, to obtain information concerning the LAN it is controlling, e.g. ACKs. These LAN agents themselves form a "higher-level" peer group. One serves as a distinguished parent node, others as subordinates. The LAN agents are communicating with each other to arrange for forwarding messages, and they jointly calculate the ACK information for the entire scope, which in

this case could be e.g. a data center in which the LANs reside. The distinguished node that hosts the parent LAN agent also hosts a yet higher-level component, call it a "data center agent". This agent could communicate with the sender, or the construction might continue further in a similar fashion. Note how in this example the peer groups defined at various levels overlap with scope boundaries.

Note also that as long as their interfaces match, each peer group could run an entirely different algorithm. We believe this power could be extremely useful in settings where local administrators control policies governing, for example, use of IP multicast and hence where different groups may need to adhere to different rules.

The issue of how to select protocols at different levels in such a way that their interfaces would match is beyond the scope of this paper. In our forthcoming paper [7], we introduce a new mechanism that could help address this issue in a more systematic manner.

To keep the presentation simple, in the model and in the examples we discussed a peer group handles recovery in a single topic. In our full design, a group of peers can handle recovery in multiple sessions at once. Throughout the lifetime of the group, peers will be instructed to begin recovery for certain sessions, at some point later they will enter the flushing phase for specific sessions (while other sessions may still be active), and may finally be requested to cease any activity for specific sessions. Accordingly, a peer, via its bottom and upper interfaces, exchanges data and requests related to multiple sessions at once. One may think of a peer as having multiple pairs of bottom and upper interfaces, each pair for a different set of sessions. Also, peers hosted at a physical node will not necessarily form a vertical, linear stack, as in our examples, the same lower-level peer may interact with two or more peers at a level above it. We omit details for clarity. All techniques that we introduced here carry over to the full design.

2.10. Implementing Recovery Algorithms

In section 2.8 we have explained how a hierarchy of recovery domains is built, such that for each session, the domains "responsible" for it form a tree. In section 2.9 we gave an example of how an algorithm such as RMTP can be modeled in our framework as a network of agents that handle the recovery tasks at various levels. A distributed recovery domain **D** in our framework will correspond to a peer group. When **D** is created at some scope **X**, the latter selects an algorithm to run in **D**, e.g. a ring or a tree, and then every sub-domain $\mathbf{D_k}$ of **D** is requested to create an agent that acts as a "peer $\mathbf{D_k}$ in group **D**". Note how the membership algorithm provides membership view at one level "above", i.e. the scope that owns a particular domain would learn about domains in all the sibling scopes. This is precisely what is required for each peer $\mathbf{D_k}$ in a group **D** to learn the membership of its group.

When the SM of a scope **X** learns that an agent should be created for one of its recovery domains $\mathbf{D_k}$ in group **D**, two things may happen. If **X** manages a single node, the agent is created locally. Otherwise, **X** delegates the task to one of its sub-scopes. As a result, the agents that serve as "peers" at the various levels are delegated to individual nodes. We thus arrive at a structure just like on Figure 18, where each node has a stack of one or more agents, each operating at a different level, linked to one another. When the node hosting a "higher-level" agent crashes, the agent is delegated to another node. Since our framework would transparently recreate channels between agents, it looks to other peers agents as if the agent lost its cached state (not permanently, for it can still query its bottom interface and talk to its peers). This requires that algorithms be defined in a way allowing peers to crash and resume with some of their state erased. Based on our experience, for a wide class of protocols this is not hard to achieve.

3. Evaluation

The need for brevity precludes a detailed discussion of the performance of our architecture. The strength of this design lies in its extensibility, ability to accommodate a wide range of transport and recovery protocols, and in facilitating the cooperation among independent parties in creating a global publish-subscribe infrastructure. Such benefits are hard to quantify. However, in certain scenarios, our approach can also greatly improve scalability. In [8], we show how we used the model and principles presented here as the basis for the design of QSM [1], a new publish-subscribe platform offering a simple ACK-based reliability and extremely scalable in multiple dimensions. We are also in the process of creating a reference implementation of the infrastructure outlined here. Ultimately, this effort will lead to a set of specifications similar to [2].

5. References

[1] K. Ostrowski, K. Birman, and A. Phanishayee, "QuickSilver Scalable Multicast". In submission, 2006.

[2] http://ifr.sap.com/ws-notification/ws-notification.pdf

[3] http://ftpna2.bea.com/pub/downloads/WS-Eventing

[4] S. Paul, and K. Sabnani, "Reliable Multicast Transport Protocol". Journal of Selected Areas in Communications (1997).

[5] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing". IEEE/ACM TONS (1996).

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". ICDCS '99.

[7] K. Ostrowski, K. Birman, "Achieving Modularity and Scalability via Typed Communication Endpoints". Forthcoming.

[8] K. Ostrowski, K. Birman, "Extensible Web Services Architecture for Notification in Large-Scale Systems (Extended Version)". Cornell University Technical Report. Forthcoming.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems

Ken Birman, Anne-Marie Kermarrec, Krzystof Ostrowski,
Marin Bertier, Danny Dolev, Robbert Van Renesse
Cornell University, Ithaca; INRIA/IRISA and IRISA/INSA, Rennes; Hebrew University, Jerusalem

Abstract

Challenges of scale have limited the development of event notification systems with strong properties, despite the urgent demand for consistency, reliability, security, and other guarantees in applications developed for sensitive tasks in large enterprises. These issues are the focus of Quicksilver, a new multicast platform targeted to large-scale deployments. An initial version of the system can support large numbers of overlapping multicast groups, high data rates and groups with large numbers of members. However, Quicksilver still requires manual help when discovering the system configuration and can't easily enforce certain types of application monitoring and integrity constraints. In this paper, we propose to extend Quicksilver by introducing gossip mechanisms, yielding a self-managed event notification platform. The two technologies are presented through a single interface and appear to end users as live distributed objects, *side-by-side with other kinds of typed components.*

1. Introduction

As we look to the next generation of distributed computing platforms, it is hard not to feel concern at the accelerating deployment of systems that will play sensitive roles, and yet will be built using fragile technologies. For example, an electronic health records system must achieve high levels of availability and consistency, be largely self-configuring, and maintain privacy and security. A typical deployment scenario involves decentralized systems linked over networks, integrating subsystems running at hospitals, other care providers, laboratories, insurance companies, pharmacies, etc. Electronic monitoring devices and other sensors running both in the hospital and at home will contribute time-sensitive data, and some therapeutic and drug delivery devices will be remotely controlled.

To reduce cost and leverage standardization, a system of this sort would probably be constructed using COTS platform technologies, such as web services. Doing so also brings productivity benefits, in the form of development tools and runtime support, and makes it easy to integrate pre-supplied functions with new application-specific ones. However, today's solutions lack the sorts of strong properties needed for sensitive uses. Our objective is to extend these platforms by adding robust tools that bridge gaps while complying with standards.

The nerve center of a modern service-oriented architecture is its event notification subsystem. Event notification services can distribute sensor readings and other kinds of updates to widely distributed system components, and can be used to replicate information where an application or a record is available at multiple locations. By decoupling publishers from subscribers, these services make it easy to upgrade an application over time and to integrate components that run on dissimilar platforms or were implemented using very different technologies. On the other hand, traditional event notification platforms lack the strong guarantees needed for medical decision making and other critical roles.

If we can create a new kind of scalable, robust event notification architecture that fits seamlessly into modern development platforms such as Windows .net or J2EE, and yet has strong properties that reduce to rigorously specified protocols that the end user can count upon and reason about, we can help application developers create robust applications for sensitive uses.

In this paper, we focus on scalability, robustness and self-management, deferring issues of security and privacy for the future. For scalable event notification with strong reliability guarantees, we've developed Quicksilver: a high-performance multicast technology that can implement a variety of reliability models, including consensus-based ones [1][2]. Traditionally, systems implementing reliable multicast have scaled poorly, but as reported below, this problem can be overcome. Moreover, although we don't tackle the question here, we believe that Quicksilver can be secured using digital certification certificates, by authenticating access to information resources, and encrypting all network traffic using per-event-channel keys that can be refreshed whenever the set of subscribers changes.

The existing version of Quicksilver is weaker with respect to self-configuration and self-management; both critical requirements for the sorts of applications we hope to support. In our target environments, the pace of reconfiguration could be very rapid: if a patient falls ill, providers might (in effect) hand the family a box full of equipment to be deployed throughout the home. Needs change as the patient's care plan evolves. Patients are moved from unit to unit. Thus one must imagine a highly dynamic, rather unpredictable environment in which the

---

[1] Contact: ken@cs.cornell.edu; The Cornell research group was supported by grants from AFRL/IFSE, AFOSR, NSF and the Intel Corporation.

sets of components, their configurations, and their communication patterns change constantly. Against this backdrop, we seek an event notification infrastructure that can configure itself, that can adapt as conditions evolve, and that can be leveraged to support self-configuring applications.

Fault-tolerance poses closely related problems. Today, Quicksilver offers fault-tolerance through models such as virtual synchrony, where applications are structured into groups and, if desired, will be notified when membership changes. But not all integrity constraints map easily to group membership tracking. For example, the decoupling of publisher from subscriber is advantageous from a development perspective, but sometimes correct function requires that there be an active subscriber associated with certain topics. One such case involves logging accesses to patient records for offline audits. If this functionality is implemented using event notification, it important that the logging service be running when audit events are published. Yet even if built upon a substrate such as Quicksilver, today's event notification APIs lack mechanisms to express such constraints, and hence can't trigger exceptions when they are violated.

To address self-* needs, both within Quicksilver and in applications built using it, we propose to use technology emerging from work on *gossip* protocols. Gossip encompasses a large class of protocols that exploit randomness to achieve surprising robustness under a wide range of operating conditions. They can be made self-configuring, adapt rapidly after disruption, and support a diversity of useful end-user functionality.

The integration of gossip with multicast in a single setting poses non-trivial systems-engineering challenges. Here, we propose such a unification. Although our new system is still under development, it will offer a seamless infrastructure in which Quicksilver runs side-by-side with gossip-based mechanisms to provide a self-managed scalable event notification capability. The system will expose these gossip mechanisms so that applications can exploit them directly in the same paradigm used to expose Quicksilver's multicast functionality. Here we sketch out the architecture and discuss some research challenges it poses; several appear to be of broader relevance.

The paper is structured as follows. First, we spend a moment discussing the strengths and limitations of gossip technologies. The goal is not to be exhaustive, but rather to identify styles of gossip that are both highly effective and well matched to our self-management objectives. Next, we review Cornell's new platform, Quicksilver, touching both on its scalability and its unusual embedding into the Windows .net framework. The latter topic emerges as a source of leverage in what we are now proposing to do. Finally, we explore the options for integrating the two, arriving at an architecture that (we believe) is interesting in several respects. First, it sets gossip side by side with scalable event notification. Next, the system offers an elegant embedding into Windows so that developers can benefit from that system's powerful component integration functionality and development tools (a Linux version is also under design). And finally, it suggests a path for future evolution of service-oriented architectures and standards. The paper concludes by discussing open research questions.

2. Gossip protocols

A *gossip protocol* is one with the following properties:

1. The core of the protocol involves periodic, pairwise, inter-process interactions.
2. The information exchanged during these interactions is of (small) bounded size.
3. When node *a* interacts with node *b*, the state of *a* evolves in a way that reflects the state of *b* (and vice versa). For example, if *a* pings *b* merely to measure RTT, this is not a gossip interaction.
4. Reliable communication is not assumed.
5. The frequency of the interactions is relatively low when compared to typical message latencies.
6. There is some form of randomness in peer selection.

There are three prevailing styles of gossip protocol.

1. *Dissemination (rumor-mongering) protocols*. These use gossip to spread information; they basically work by flooding nodes in the network, but in a manner that produces bounded worst-case loads:
   a. An *event dissemination* protocol runs in response to events and can be understood as using gossip to carry out multicasts, although the events don't actually trigger the gossip (since gossip runs periodically).
   b. A *background data dissemination* protocol gossips continuously to track the evolution of state at participating nodes.
2. *Anti-entropy protocols* repair replicated data by comparing replicas and reconciling differences.
3. *Aggregation protocols* compute a network-wide aggregate by sampling information at the nodes in the network and combining the values to arrive at a system-wide value – the number of nodes in the system, the sum or average of some value, etc

Our definitions are rather broad; indeed, many protocols that predate the earliest use of the term "gossip" fall within our definition. In particular, notice that a gossip substrate can "mimic" a standard routed network. That is, nodes could "gossip" about traditional point-to-point messages, in effect tunneling normal traffic through a gossip layer. Bandwidth permitting, this implies that a gossip system can potentially support any classic protocol or distributed service. Nonetheless, when we talk of gossip, we rarely intend such a broadly inclusive

interpretation. More typically we have in mind protocols that run in a regular, periodic, relatively lazy, symmetric and decentralized manner; the high degree of symmetry among nodes is particularly characteristic. To illustrate this point, consider that one could run a 2-phase commit protocol over a gossip substrate, piggybacking the messages on gossip traffic. In our view, doing so would be at odds with the spirit of the definition: there's nothing wrong with such a protocol, but it isn't gossip!

2.1 The Limitations of Gossip

The stylized manner in which we normally use gossip introduces significant limitations. First, consider the implications of the small, bounded message sizes and the relatively slow periodic message exchanges. These combine to limit the information carrying capacity of a gossip algorithm. For example, if gossip is used to disseminate information (often, in a form of flooding), the system-wide capacity for new events will be limited simply because the aggregate "bandwidth" available is bounded. The problem is that gossip protocols keep the nodes in a network busy while information spreads – typically, a process that requires O(log(n)) time. It follows that the "rate" at which events can be introduced will be proportional to 1/log(n).

The relatively slow spread of gossip can also be an obstacle. While it is common to claim that users need only tune the gossip rate to match their goals, requirement 5 complicates the picture. Gossip rates approaching the network RTT are out of the question.

Finally, gossip can be fragile in the face of malicious behavior (components that malfunction, for example by running the protocol incorrectly, disseminating incorrect data, and so forth). Recent work on BAR Gossip [21] tries to overcome some of the issues by using verifiable pseudo-random peer selection to avoid selfish and malicious behaviors. But this is just a first step.

2.2 Strengths of Gossip

Although gossip has limitations, these protocols do have substantial power. Among the most cited strengths are these:

• *Convergent consistency.* Properly designed gossip protocols, when not overwhelmed by a higher rate of incoming "events" than the information-carrying bandwidth of the underlying channels, should have a logarithmic mixing time – any new event will, with high probability, affect all nodes that need to learn about it within time logarithmic in the system size.

• *Emergent structure.* Earlier, we contrasted a classic deterministic protocol for building a spanning tree by leader-initiated flooding with a decentralized way of building such a tree using gossip. In the gossip style, the tree "emerges" from randomized pairwise interactions between peers. The term emergent structure is intended to evoke the image of a data structure that emerges with probability 1.0 in this manner. The structure may then continue to evolve over time as further gossip occurs.

• *Simplicity.* Most (but not all) gossip protocols are extremely simple and highly symmetric, with all participants running the same code.

• *Bounded load on participants.* Many classic (non-gossip) distributed protocols are criticized because they can generate high surge loads that overload individual components. Gossip is normally used in ways that produce strictly bounded worst-case loads on each component, eliminating the risk of disruptive load surges. In some situations, where network capacity is also a concern, peer-selection is further biased to control load imposed on network links.

• *Topology independence.* If running on a sufficiently connected networking substrate, and with sufficient bandwidth, a gossip protocol will often operate correctly on a great variety of underlying topologies.

• *Ease of local information discovery.* Many gossip protocols are used for purposes of discovery, for example to find a nearby resource (these are usually protocols in which gossip occurs between neighbors, not between arbitrarily distant peers). Unlike local flooding, which scales poorly, gossip would typically find local information less quickly but with bounded costs: perhaps, a constant or a delay logarithmic in the system size.

• *Robustness to transient network disruptions.* As time elapses, there are exponentially many routes by which information can flow from its source to its destinations. However, not all uses of gossip are robust in all ways. For example, unless data is self-verifying, dissemination protocols are often vulnerable to data corruption. Anti-entropy protocols may similarly be at risk if a replica becomes corrupted. And aggregation protocols are vulnerable not just to the introduction of faulty information, but also to computational errors that result in a faulty computation of the aggregate.

2.3 Appropriate roles for gossip

The foregoing discussion suggests a number of natural roles for gossip in large-scale event notification systems.

The earliest uses of gossip were to disseminate information in large-scale systems [22]. Scalability and robustness were cited as the primary benefits in these uses: the load on each node grows in a logarithmic manner as the system scales and information can be

reliability disseminated in the presence of a high proportion of node failures [20]. Such properties rely on the fact that each node samples network state randomly. This pseudo-randomness can nonetheless be controlled or "shaped". For example, Lpbcast [19] and Cyclon [15] are protocols in which each peer periodically selects another peer with which it gossips; they differ in the details of target selection, and in the way they merge information gathered through the gossip exchange with their own.

Generalizing these ideas, gossip may be used to create unstructured overlay networks, achieving properties close to those of random graphs [12]. Having used gossip to create such a graph, gossip protocols can also run over them, for example to create an overlay optimized with respect to an application-specific metric. For example, T-man builds overlays that use application-supplied quality functions to bias neighbor selection [10]. In [14], the gossip itself is biased; users with shared interests are structured into peer groups for file sharing, substantially improving response times in a search application.

Similarly, GosSkip [17] and Sub-2-Sub [13] build content-based publish-subscribe systems in which the overlay topology matches the subscription pattern. In GosSkip, subscriptions are organized into a skiplist structure so that events will be routed to interested subscribers in a logarithmic number of hops. In Sub-2-Sub, several gossip-protocols are layered to efficiently support range subscriptions. The lowest layer uses random peer sampling to ensure connectivity and robustness, a second layer creates clusters of "close" subscriptions, and the third layer structures overlapping subscriptions to ensure an exact and exhaustive dissemination of events.

This flexibility comes at a price. Gossip-based publish-subscribe overlays are often slow: the technology is wonderful for matching publishers with subscribers, but says little about getting events delivered rapidly, robustly, and with strong reliability properties. Indeed, we like to think of these kinds of applications as having two disjoint aspects: a gossip infrastructure that, in these cases, builds an overlay; and then a distinct dissemination structure that uses the overlay to reliably distribute events.

This way of thinking leads back to our current goals. We hope to systematically ask how gossip can be valuable in event-notification systems such as Quicksilver and in the applications that run over it. A number of options seem to be worth exploring. For example, as just seen, a gossip-constructed overlay network could be useful for efficient dissemination. In this case, Quicksilver itself would provide the "quality metrics" used to optimize the overlay, and the associated cost functions would reflect the mechanisms Quicksilver uses for dissemination and for recovery of lost packets.

More broadly, we hope to use gossip to materialize a form of distributed "picture" of the application network, which would become an input to an auto-configuration application that would generate configuration files. These would advise the end-user application (in addition to the Quicksilver event notification infrastructure) of the topology on which it should operate and the appropriate parameter settings to use. Later, as conditions evolve, the same approach could be used to reconfigure the running system so as to repair damage caused by a failure, or to integrate new components with the existing infrastructure.

Another possible role for gossip would be to track overall loads, loss rates and other status in the system. We have experience with a gossip-based system used for this purpose. Astrolabe is a distributed monitoring and data mining system that uses gossip to construct a virtual hierarchical database that can be queried much like a normal database [5]. The database is extremely useful for self-optimization and problem diagnosis. Because Astrolabe is fully replicated it has no single point of failure or load-related hot-spots, and the underlying gossip protocol remains robust even under stress that can shut down most other system functionality. In our new system, we believe aggregation mechanisms can play even more roles, including parameter setting and dynamic adaptation [11]. Aggregation can even be used for resource allocation, for example by using gossip to sort peers according to an application-specific metric [16].

Finally, we will use gossip to support background diffusion of system information that won't be needed immediately, but could be of high value "later". A tool permitting discovery of available information sources would be one possible use for such a mechanism. Other possibilities include mechanisms for tracking contact nodes or other services, finding information stored elsewhere in the network, etc. By using gossip to disseminate the underlying information, we can be certain that data will get through even if the system configuration changes (or is disrupted), and hence will be available when and where needed.

To exploit these kinds of gossip mechanisms, we need to tackle some significant software engineering issues that prior work has largely overlooked. To make gossip useful as a tool, one needs appropriate embeddings of these abstractions into the runtime environment. For these purposes, we propose to extend a feature of Cornell's Quicksilver platform, discussed below.

3. Quicksilver

Cornell's Quicksilver project [3][4] offers a scalable event notification infrastructure that can support strong properties on a per-topic basis. An application can subscribe to large numbers of communication channels, with the properties of each channel matched to the data it carries. Krzysztof Ostrowski is the lead architect and developer for Quicksilver, in collaboration with Ken Birman, Danny Dolev and Robbert van Renesse. We start

by reviewing prior work on Quicksilver, and then suggest some of the extensions our new effort will explore.

A key objective for Quicksilver is scalability in multiple dimensions: numbers of applications using the platform, numbers of event channels to which each application subscribes, data rates, tolerance of disruption, etc. Our underlying premise is that inadequate scalability has limited the uptake of group-multicast in general, and has prevented its widespread use in support of event notification. This sometimes manifests itself through throughput that degrades gracefully as the system is deployed into a larger setting, but more dramatic consequences are also observed. For example, many large-scale event notification platforms become unstable in large deployments, oscillating from very low throughput to overwhelmingly high data rates in which traffic generated by the platform can actually shut down the communications bus by swamping it with data, retransmissions, nack and ack messages and other forms of overhead – a so called broadcast storm effect. In designing Quicksilver, our goal was to demonstrate stability in this problematic domain.

This is not the right setting for a detailed discussion of the Quicksilver architecture. Instead, we summarize some key ideas very briefly:

- *Separation of concerns.* Quicksilver treats event dissemination separately from recovery of lost packets, flow control, and imp lementation of stronger consistency ("properties").
- *Regions of overlap.* A single node will often subscribe to many event channels. If each channel is treated as a separate multicast group, one encounters obvious problems of scale. Accordingly, Quicksilver maps from overlapping channels down to *regions*, defined to be sets of nodes with similar subscriptions. Dissemination is on a per-region basis; recovery is done in an aggregated manner over regions, etc.
- *Scalable recovery.* Quicksilver uses a novel hierarchy of token rings to achieve scalable detection of lost packets and, when possible, to recover data between peers in a region, offloading work from the sender.
- *Per-channel reliability properties.* The reliability properties of each channel can be matched to its role.
- *Managed runtime environment.* Quicksilver runs in managed settings, allowing it to leverage strong type checking, memory management, etc.

Details of the architecture and protocols appear in [3][4].

Quicksilver has been running since June 2006. For the moment, all our users are building datacenters – WAN scenarios are a goal once the new gossip-based mechanisms are available, but the current system doesn't run in WAN settings. In our datacenter experiments, we've set up groups with up to 200 nodes (larger runs are planned), than subjected them to extremely high throughputs and injected various forms of stress.

Up to the present, we have seen only minimal throughput degradation and no signs of instability or throughput fluctuations even in the largest configurations. In contrast, such problems are easy to provoke in most existing technologies for multicast in the same settings, even with much smaller groups of just 50 to 75 members [2]. Quicksilver can saturate a 100Mbit ethernet interconnect with just 20-40% CPU loads on the inexpensive PC's making up our test cluster; experiments with our prior systems peaked at about a tenth these data rates and generated much heavier loads. Perhaps most important, processes are able to access large numbers of groups. For this reason, when used to support event notification, Quicksilver can maintain steady performance even when each process joins as many as 8000 separate event channels [3][4]. Obviously, this capsule summary oversimplifies in some important ways (in particular, not all configurations of processes and event streams are supported), but they do give a sense of what the system should be able to achieve.

Of primary relevance here is the manner in which Quicksilver embeds event notification channels into Windows. Traditionally, event notification platforms have been treated as a free-standing technology that lives separately from the operating system. Quicksilver can be used this way too, through a conventional publish-subscribe infrastructure that generalizes the web services eventing standards (in [6] we discuss our reasons for extending these standards rather than working entirely within ws-notification or ws-eventing).

But Quicksilver also offers a second, deeper embedding into Windows in which event notification channels can be accessed either as a new kind of distributed *live object* visible in the file system side-by-side with other named objects. These objects are best understood as distributed abstract data types. A program accesses such an object much as it would access a file in Windows: given appropriate permissions, it can open the object, read the current state, and will receive events as the state is subsequently updated. This, however, is an illusion: the "object" is really an event channel, and the state is a checkpoint produced by some existing subscriber when a new program subscribes. State persistence is available, but optional.

We've emphasized the similarity between the way that a system such as Windows understands file "types" as an association between the data in some object and the programs that implement operations on that kind of object, and the way that Quicksilver associates a type with each event notification channel. For Quicksilver, the type corresponds to an object class, but also is associated with a definition of the properties the channel should implement. The effect is to confer a distributed semantics on the group of objects as a whole. The approach is flexible enough to support weak properties such as best-effort notification, stronger consensus-based properties

such as the virtual synchrony model, or even very strong models such as transactional 1-copy serializability. Quicksilver implements a domain-specific programming language within which the properties associated with each event channel can be specified. The system basically compiles these property definitions into pseudo-code which it can execute to achieve the desired behavior.

4. A unified platform

For our purposes, the key point of leverage involves the embedding of Quicksilver's live objects (event channels) into Windows. Consider the integration of abstract data types such as Excel spreadsheets or Word documents into the Windows file system. Windows uses the filename extension to understand the "type" of the object, allowing it to interpret operations on the object as method invocations on an appropriate application program. Web services standards are used in conjunction with these componentization mechanisms: active components such as the Excel application register their interfaces using the Web Services framework built into .net, at which point the Windows platform can function as a component integration environment using Web services standards and protocols to perform tasks such as method invocation. Of course, this component-to-component type system is somewhat primitive, but one could imagine taking the idea much further; indeed, there are projects underway at Microsoft to do just that. It isn't unreasonable to imagine that future versions of Windows will incorporate a full-fledged distributed type system at the component level.

As suggested above, Quicksilver extends Windows to support abstract data types with "live" content, and allows a variety of event stream providers to support the live aspects of the abstraction. A Quicksilver event notification channel has a name that can be visible in the file system name space, and a type, corresponding to the properties associated with the event channel. When an application binds itself to an event channel, Windows passes the binding event to Quicksilver, and we can perform type compatibility checking, or can even perform some kinds of dynamic type coercion (for example by introducing an encryption/decryption layer in order to integrate a component that doesn't support encryption with an event channel that requires stronger forms of security). The same mechanisms also work from the Windows shell: if a user right-clicks on a Quicksilver event channel, the shell extensions framework passes us the request. Quicksilver can then identify applications that can connect to this kind of channel, and can even generate dynamically created virtual folders, for example displaying thumbnail-size images from a video streaming application.

Quicksilver is thus on a path towards the same kind of tight integration with Quicksilver event streams as is seen with other Windows communications options such as DCOM. The approach enables developers to leverage existing Windows application development and debugging tools while benefiting from co-existence in a managed framework. If Windows evolves in the manner currently anticipated, type checking will become possible even across component boundaries. Because Quicksilver uses the CLR memory management layer, no copying occurs when a large object is multicast. Of course, such a positioning of the technology also brings challenges of its own (for example, to maximize performance in a managed environment requires protocol designs quite different from those one uses in a Linux/C multicast implementation [3]) but the problems are solvable and we believe the result is well worth the effort. We should comment that although Windows is our initial target, everything we are doing should port (using Mono) to Linux and would then be accessible from J2EE or even Corba applications.

This, then, is the core contribution of the present paper: a vision of how one might unify these three worlds: objects in a platform such as Windows on the one hand, and both gossip and of scalable event notification on the other, all in a single framework. A first step towards this vision requires that the Quicksilver multicast framework be separated from the mechanisms that embed Quicksilver objects into Windows; Ostrowski is already developing this capability as part of version 2.0 of the system. As is the case in Quicksilver today, the basic abstraction will be that of a distributed object having a "state" and an associated event stream. However, rather than assuming that the live content is transported by Quicksilver's reliable multicast protocols, there will be at least two possible communication infrastructures – the other being gossip-based. Down the road one might imagine additional options, such as an IP-TV streaming layer, or one focused on real-time communication.

Thus, referring back to the examples of gossip-based mechanisms mentioned in Section 3, one could build a gossip-based topology and configuration discovery service that, in effect, produces an annotated picture of the state of the system. An end-user could access that picture by clicking on an associated file name; doing so would launch some sort of browser capable of visualizing this kind of information and might let the user explore the network, for example to pin down a bottleneck that is impacting performance. Application programs could use the picture to configure themselves. And Quicksilver's event notification infrastructure could use that picture to construct overlays for disseminating events that use IP multicast when possible, but tunnel data through overlay trees where IP multicast is not feasible (and these same overlay networks would also be available to application designers, through some form of abstract data type). The remarkable robustness of the gossip protocols ensures that even when all else is disrupted, applications can still

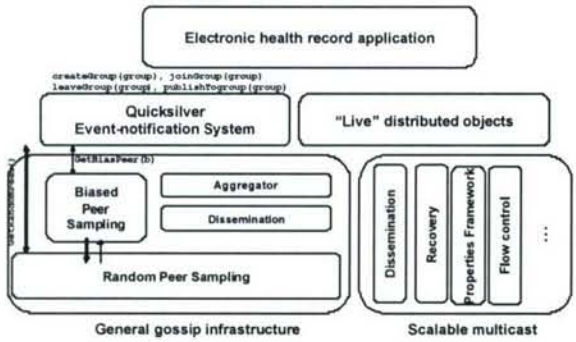


Figure 1: Overall System Architecture

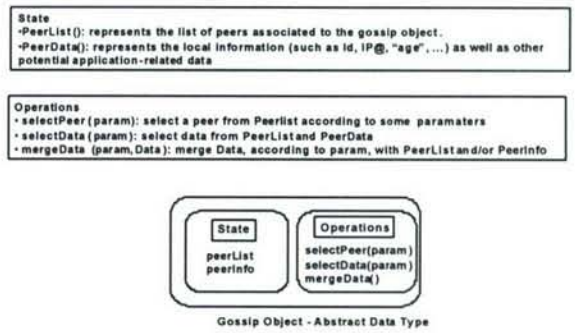


Figure 2: Generalized Implementation of a Gossip Object

monitor the system to set parameters, configure themselves, and adapt when conditions change.

But we believe we can do more than to simply import gossip functionality into Quicksilver. Gossip systems of the types we reviewed share substantial commonalities across their various presentations. For example, many gossip mechanisms require random peer selection, either within the full system (a kind of *anycast*) or within a set of neighbors of a node (a local variant on *anycast*). The thinking is that this and other low-level primitives can be standardized within the gossip subsystem, and then reused across gossip-based objects. Doing so poses interesting research challenges: if a single object employs *anycast*, one can implement a "greedy" solution. But suppose that on some single node there are tens or even hundreds of gossip-based objects, all using *anycast*. Could we aggregate, so that a single message can carry information on behalf of multiple objects?

One can pose similar questions at a higher level. Many gossip algorithms are highly stylized: the nature of a gossip exchange is rather similar across most gossip-based mechanisms, even if the details of what "state" is exchanged and how it is "merged" differ. This immediately suggests that one might design an abstract gossip state-machine that could be instantiated in multiple objects, parameterized with appropriate state marshalling and merge functions.

The resulting architecture is summarized in Figures 1 and 2. Figure 1 illustrates the overall system architecture, with the gossip infrastructure hosted side-by-side with the scalable multicast infrastructure and accessed either through a generalized publish-subscribe interface, or in the form of live distributed objects. As noted earlier, internal details for Quicksilver can be found in [3] and will not be repeated here. Figure 2 gives some additional detail for the gossip infrastructure.

5. Electronic health record example

We conclude the discussion by revisiting our electronic health record example, assuming now that the gossip mechanisms and the Quicksilver-based event notification solution are available side-by-side.

Let's start with roles for the gossip mechanisms. For the time being, we've decided to focus on uses in which the gossip components will be simple enough so that we can verify correctness, able to "sanity check" data collected from the environment, and unlikely to come under attack; these assumptions mitigate the security concerns mentioned earlier. For example, with gossip it isn't difficult to build a system that can track locations of system components: servers, client platforms, sensors, other devices. When a change occurs, the updated configuration should become visible with delay proportional to the log of the size of the system – in the scenarios we have in mind case, probably within 10 or 15 rounds of gossip. This capability could be the basis for a highly robust plug-and-play technology, whereby the health-care system would adapt in tens of seconds as conditions evolve. Although such a system might collect incorrect information about a platform that has some form of scrambled configuration state, the "damage" would be limited to the annotation of that component on the map, and the gossip objects can be designed to sense and reject implausible inputs.

Gossip could also be used to monitor system invariants (such as: "there should always be at least one instance of the auditing service"). Here, Quicksilver's notion of membership offers very rapid event detection and reaction, but if enough damage occurs while the system is running to seriously disrupt event notification, the gossip layer could guide a timely discovery of the problem and dynamic repair or adjustment of the parameters. The remarkable robustness of gossip mechanisms gives us reason for confidence that they will be able to continue to operate reliably even when other infrastructure components are severely degraded by a disruptive event.

Gossip can also be used to help system components connect themselves in appropriate ways. For example, a component might keep track of the locations of the various servers so that in the event of a fault that prevents

connection to one server, the clients using it can seamlessly roll over to others offering backup functionality. When the first server recovers, the clients can shift back. Gossip mechanisms can be used to monitor system health, assisting managers in diagnosing and repairing problems that arise because of software bugs or other disruptive events. If a firewall or server comes under attack (or just becomes overloaded), gossip based tracking mechanisms can help client systems discover the problem, identify fall-back options, and gracefully adapt.

Gossip also offers an antidote to certain kinds of fragility. For example, suppose that we want to track the physical location of patients in our hospital complex. In the most obvious standard implementation of an electronic health record system, one would probably place some sort of active component on the patient's gown or bed; it would continuously track its own location (somehow) and report that data to the central database. With gossip, new and potentially more robust options arise. Now, client systems can gossip with one-another about patient "sightings". With many observers and many paths by which information can spread, we obtain a patient location-tracking database at low cost, and with guarantees of extremely robust behavior even in the event of a disruptive condition, such as a malfunctioning application that generates extremely high network loads and loss rates. (Recall from our discussion of Astrolabe that a gossip management infrastructure might help in this case too, by assisting the system administrator in localizing the problem).

What about high-speed event notification and streaming? Our system could exploit this functionality in a great many ways. If we assume that health care records are, in effect, replicated throughout the system as a whole, when an update occurs, it will be important to consistently update all copies. Here we see a form of event notification that requires relatively strong reliability and delivery semantics – corresponding to a consensus-based model such as virtual synchrony or state machine replication, both available within Quicksilver as group "types". Event notification can support a publish-subscribe relationship between the database servers in the hospital and client systems operated in private practices and other satellite locations. Bedside or nursing station display systems may need to be refreshed. Similarly, if the update is relevant to a patient's prescriptions, the event might be pushed out to participating pharmacies. One can also imagine high-throughput event channels. For example, television cameras and other sensors monitoring infants in a neo-natal unit could stream images to the nursing station; pediatricians would be able to subscribe as necessary to keep an eye on their patients: a robust, scalable IP-TV architecture

The Quicksilver properties mechanisms would be beneficial here, by permitting the system to match the properties of each type of event channel, or live object, to the requirements associated with that category of object. In fact we doubt that there would be a huge number of cases, but there are clearly subsystems that would value real-time data delivery over other guarantees, subsystems that need the sorts of consistency afforded by virtual synchrony or state machine replication, and subsystems that need transactional "ACID" properties. These can all be supported, side-by-side, on a per-event-channel basis.

These example illustrates a point worth reiterating:b y using the publish-subscribe paradigm, the publishing side of the enterprise can be designed independently from the data consuming side; both can be incrementally extended over time as new applications are added, and will automatically accommodate varying runtime configurations. In effect, we are able to separate the information representation standards used within the system (including the hierarchy of topics) from the data sources and the data consumers. The communications infrastructure provides the needed guarantees, and when a new component is introduced, existing event-generating applications don't need to be modified. Because Quicksilver has a strong notion of types associated with event channels and live objects, we can do far more type checking than is traditionally feasible in publish-subscribe settings. For example, we can potentially ensure that the properties of a channel match the expectations of the application that binds itself to that channel. Moreover, to the extent that we need instant detection and reaction to a failure, because Quicksilver extends the publish-subscribe eventing model to also offer (optional) information about subscription changes when processes join and leave a channel, all sorts of rapid fault-tolerance mechanisms can be implemented.

We've avoided discussion of privacy and security issues, despite their central importance in electronic health care systems. This is in part because Quicksilver currently lacks a comprehensive security architecture, although we do have some ideas for how we might build one. Our thinking is to focus on capabilities enabled by the secure replication of security keys using the algorithms of Reiter [8][9] or Rodeh [7]; these offer ways to refresh keys when the set of nodes in the replication group (the event channel) changes because of a failure or a join. However, prior research has never explored scalability implications of these kinds of secure key replication schemes, and we believe the topic will require a substantial research effort to fully resolve. Use of security keys in gossip settings represents an additional intriguing option for study.

7. Research topics

Our vision raises a number of questions:

1. Given a proposed large-scale application, what is the most effective development methodology for mapping it down to application-specific functionality, as opposed to platform-supplied functionality? How should the developer make decisions concerning the aspects that are best matched to gossip communication, those best matched to event notification, and those that require hand-coded logic? Given that both gossip and event notification systems can support "guaranteed" properties, how should the developer decide which properties are needed by a given application, and how best to achieve them? Is there are large-scale methodology for specification of overall properties of a complex system that might lend itself to a formal verification process analogous to the ones used to reason about and ultimately prove correctness for non-distributed systems? Can the properties mechanisms used in Quicksilver today be extended to include gossip protocols?
2. If a single computer system supports multiple "live" data objects, high performance often requires that protocols be designed to amortize costs. Much of the innovation in Quicksilver is at this level: the system looks for ways to disseminate data, recover from packet loss and control data rates that are aggregated across potentially huge numbers of objects. When we introduce new classes of objects supported by gossip, the gossip infrastructure will need to address similar questions.
3. We alluded to the need to secure the platform, and to the risk that gossip mechanisms might be incapacitated by certain kinds of malicious behaviors. Our architecture poses significant opportunities for research on security, ranging from questions of precisely how one might secure a gossip protocol to broader issues of scalability that arise if an application subscribes to a large number of secured objects. How should one secure a high-speed event channel? What issues arise as one scales a security abstraction in a setting where each separate event channel or live object might have its own security requirements?
4. The creation of appropriate abstractions for the gossip infrastructure is an important challenge. At the lowest level, one imagines mechanisms for random peer selection, state exchange and merge, aggregation, etc. Ideally, these should be highly standardized. Yet some gossip protocols bias peer selection, implement "tricky" state exchange/merge mechanisms, or perform aggregation in unusual ways. Needed is a platform that can function well as a black box, and yet that can also expose functionality as needed.
5. We need to better understand the correct set of gossip mechanisms needed for purposes of self-management and self-configuration in Quicksilver. The modern internet is complex, and while it is easy to evoke a vision of an autonomic infrastructure that can support plug-and-play behavior in almost arbitrary settings, implementing that vision is quite a different matter.
6. Applications running on the event notification infrastructure will also need self-management and self-configuration functionality. Quicksilver's needs are somewhat peculiar to its role; will the same autonomic mechanisms that work for Quicksilver be adequate for other purposes, or are other kinds of gossip tools needed?
7. Obtaining high performance in large-scale settings that involve managed frameworks (C# in .net, in our case) is surprisingly hard [3]. It is likely that we will need to overcome similar challenges as we implement a gossip-based infrastructure and then tune it to cooperate cleanly with Quicksilver.
8. We commented that one key to scalability in Quicksilver is the mapping of event channels down to regions of approximate overlap – sets of nodes with similar subscription sets. A basic assumption underlying the system is that this can actually be done and that large systems will exhibit high degrees of overlap, or at least that they can be designed to have this property. But how can overlap regions be discovered in the first place? We are thinking that gossip mechanisms could be very useful in discovering applications and their "potential" subscription sets, enabling an offline analysis (perhaps with a human designer in the loop) to identify regions of overlap and configure Quicksilver. In contrast, the alternative of trying to discover regions at runtime by analysis of subscription patterns as programs come and go raises a number of thorny problems and may not be the best approach.

9. Conclusions

Scalable event notification systems capable of offering strong properties may be the key to enabling a new generation of trustworthy distributed applications, but only if they can be integrated naturally into the most powerful development environments and made autonomic: self-monitoring, self-configuring, and self-managing. For these latter purposes, we propose to build a new kind of distributed abstraction that embeds into Windows much like a typed object, but can be supported either by Quicksilver's scalable event architecture or by gossip-based protocols. A system realizing this vision is now under joint development at IRISA/INRIA in Rennes and at Cornell University.

7. References

[1] Reliable Distributed Systems Technologies, Web Services, and Applications. Birman, Kenneth P.

2005, XXXVI, 668 p. 145 illus., Hardcover ISBN: 0-387-21509-3

[2] A Review of Experiences with Reliable Multicast. K. P. Birman Software Practice and Experience Vol. 29, No. 9, pp, 741-774, July 1999

[3] Implementing Scalable Publish-Subscribe in a Managed Environment. Krzysztof Ostrowski, Ken Birman. In Submission (November, 2006).

[4] QuickSilver Scalable Multicast. Krzysztof Ostrowski, Ken Birman, and Amar Phanishayee. Cornell University Technical Report TR2006-2063 (April, 2006).

[5] Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. Robbert van Renesse, Kenneth Birman and Werner Vogels. ACM Transactions on Computer Systems, May 2003, Vol.21, No. 2, pp 164-206

[6] Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. Submitted to International Journal of Web Services Research.

[7] The Architecture and Performance of the Security Protocols in the Ensemble Group Communication System. Ohad Rodeh, Ken Birman, Danny Dolev. Journal of ACM Transactions on Information Systems and Security (TISSEC). Vol. 4, No 3, pp 289-319, Aug 2001

[8] A Security Architecture for Fault-Tolerant Systems. Michael K. Reiter, Kenneth P. Birman, Robbert van Renesse. ACM Trans. Comput. Syst. 12(4): 340-371 (1994)

[9] How to Securely Replicate Services. Michael K. Reiter, Kenneth P. Birman. ACM Trans. Program. Lang. Syst. 16(3): 986-1009 (1994)

[10] T-Man: Gossip-based overlay topology management. Mark Jelasity and Ozalp Babaoglu. In ESOA 2005, Revised Selected Papers, vol 3910 of LNCS, 1-15.

[11] Gossip-based aggregation in large dynamic networks. Mark Jelasity, Alberto Montresor and Ozalp Babaoglu. ACM Transactions on Computer Systems, 23(3): 219-252, August 2005.

[12] The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. Mark Jelasity, Rashid Guerraoui, Anne-Marie Kermarrec, Marteen van Steen. Middleware 2004, volume 3231 of LNCS, 79-98, Springer-Verlag, 2004.

[13] Sub-2-Sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. Spryros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec and Maarten van Steen. Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS), Santa-Barbara, CA, February 2006.

[14] Epidemic-style Management of Semantic Overlays for Content-based Searching. Spyros Voulgaris and Maarten van Steen, Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), Lisbon, Portugal, August 2005.

[15] CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Spyros Voulgaris, Daniela Gavidia, Maarten van Steen. Journal of Network and Systems Management, vol. 13(2):197-217.

[16] Ordered Slicing of Very Large-Scale Overlay Networks. Mark Jelasity and Anne-Marie Kermarrec. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, 2006.

[17] GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles. Rachid Guerraoui, Sidath Handurukande, Kevin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant and Etienne Riviere. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, September, 2006.

[18] From Epidemics to Distributed Computing. Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. IEEE Computer, 37(5):60-67, May 2004.

[19] Lightweight Probabilistic Broadcast. Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. ACM Transaction on Computer Systems, 21(4), November 2003.

[20] Probabilistic Reliable Dissemination in Large-Scale Systems. Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. IEEE Transactions on Parallel and Distributed Systems, 14(3), March 2003.

[21] BAR Gossip. Harry Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, Mike Dahlin, Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI), Nov 2006 .

[22] Epidemic algorithms for replicated database maintenance. Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry. ACM SIGOPS Operating Systems Review Volume 22 , Issue 1 (Jan., 1988), 8 - 32

Ricochet: Lateral Error Correction for Time-Critical Multicast

Mahesh Balakrishnan[†], Ken Birman[†], Amar Phanishayee[‡], Stefan Pleisch[†]
[†]*Cornell University and* [‡]*Carnegie Mellon University*
{*mahesh,ken,pleisch*}*@cs.cornell.edu, amarp+@cs.cmu.edu*

Abstract

Ricochet is a low-latency reliable multicast protocol designed for time-critical clustered applications. It uses IP Multicast to transmit data and recovers from packet loss in end-hosts using Lateral Error Correction (LEC), a novel repair mechanism in which XORs are exchanged between receivers and combined across overlapping groups. In datacenters and clusters, application needs frequently dictate large numbers of fine-grained overlapping multicast groups. Existing multicast reliability schemes scale poorly in such settings, providing latency of packet recovery that depends inversely on the data rate within a single group: the lower the data rate, the longer it takes to recover lost packets. LEC is insensitive to the rate of data in any one group and allows each node to split its bandwidth between hundreds to thousands of fine-grained multicast groups without sacrificing timely packet recovery. As a result, Ricochet provides developers with a scalable, reliable and fast multicast primitive to layer under high-level abstractions such as publish-subscribe, group communication and replicated service/object infrastructures. We evaluate Ricochet on a 64-node cluster with up to 1024 groups per node: under various loss rates, it recovers almost all packets using LEC in tens of milliseconds and the remainder with reactive traffic within 200 milliseconds.

1 Introduction

Clusters and datacenters play an increasingly important role in the contemporary computing spectrum, providing back-end computing and storage for a wide range of applications. The modern datacenter is typically composed of hundreds to thousands of inexpensive commodity blade-servers, networked via fast, dedicated interconnects. The software stack running on a single blade-server is a brew of off-the-shelf software: commercial operating systems, proprietary middleware, managed run-time environments and virtual machines, all standardized to reduce complexity and mitigate maintenance costs.

The last decade has seen the migration of time-critical applications to commodity clusters. Application domains ranging from computational finance to air-traffic control and military communication have been driven by scalability and cost concerns to abandon traditional real-time environments for COTS datacenters. In the process, they give up conservative - and arguably unnecessary - guarantees of real-time performance for the promise of massive scalability and multiple nines of timely availability, all at a fraction of the running cost. Delivering on this promise within expanding and increasingly complex datacenters is a non-trivial task, and a wealth of commercial technology has emerged to support clustered applications.

At the heart of commercial datacenter software is *reliable multicast* — used by publish-subscribe and data distribution layers [5, 7] to spread data through clusters at high speeds, by clustered application servers [1, 4, 3] to communicate state, updates and heartbeats between server instances, and by distributed caching infrastructures [2, 6] to rapidly update cached data. The multicast technology used in contemporary industrial products is derivative of protocols developed by academic researchers over the last two decades, aimed at scaling metrics like throughput or latency across dimensions as varied as group size [10, 17], numbers of senders [9], node and network heterogeneity [12], or geographical and routing distance [18, 21]. However, these protocols were primarily designed to extend the reach of multicast to massive networks; they are not optimized for the failure modes of datacenters and may be unstable, inefficient and ineffective when retrofitted to clustered settings. Crucially, they are not designed to cope with the unique scalability demands of time-critical fault-tolerant applications.

We posit that a vital dimension of scalability for clustered applications is the *number of groups* in the system. All the uses of multicast mentioned above induce large numbers of overlapping groups. For example, a computational finance calculator that uses a topic-based pub-sub system to subscribe to a fraction of the equities on the stock market will end up belonging in many multicast groups. Multiple such applications within a datacenter - each subscribing to different sets of equities - can result in arbitrary patterns of group overlap. Similarly, data caching or replication at fine granularity can result in a single node hosting many data items. Replication driven by high-level objectives such as locality, load-balancing or fault-tolerance can lead to distinct overlapping replica sets - and hence, multicast groups - for each item.

In this paper, we propose Ricochet, a time-critical re-

liable multicast protocol designed to perform well in the multicast patterns induced by clustered applications. Ricochet uses IP Multicast [15] to transmit data and recovers lost packets using *Lateral Error Correction* (LEC), a novel error correction mechanism in which XOR repair packets are probabilistically exchanged between receivers and combined across overlapping multicast groups. The latency of loss recovery in LEC depends inversely on the aggregate rate of data in the system, rather than the rate in any one group. It performs equally well in any arbitrary configuration and cardinality of group overlap, allowing Ricochet to scale to massive numbers of groups while retaining the best characteristics of state-of-the-art multicast technology: even distribution of responsibility among receivers, insensitivity to group size, stable proactive overhead and graceful degradation of performance in the face of increasing loss rates.

1.1 Contributions

- We argue that a critical dimension of scalability for multicast in clustered settings is the number of groups in the system.
- We show that existing reliable multicast protocols have recovery latency characteristics that are inversely dependent on the data rate in a group, and do not perform well when each node is in many low-rate multicast groups.
- We propose Lateral Error Correction, a new reliability mechanism that allows packet recovery latency to be independent of per-group data rate by intelligently combining the repair traffic of multiple groups. We describe the design and implementation of Ricochet, a reliable multicast protocol that uses LEC to achieve massive scalability in the number of groups in the system.
- We extensively evaluate the Ricochet implementation on a 64-node cluster, showing that it performs well with different loss rates, tolerates bursty loss patterns, and is relatively insensitive to grouping patterns and overlaps - providing recovery characteristics that degrade gracefully with the number of groups in the system, as well as other conventional dimensions of scalability.

2 System Model

We consider patterns of multicast usage where each node is in many different groups of small to medium size (10 to 50 nodes). Following the IP Multicast model, a group is defined as a set of receivers for multicast data, and senders do not have to belong to the group to send to it. We expect each node to receive data from a large set of distinct senders, across all the groups it belongs to.

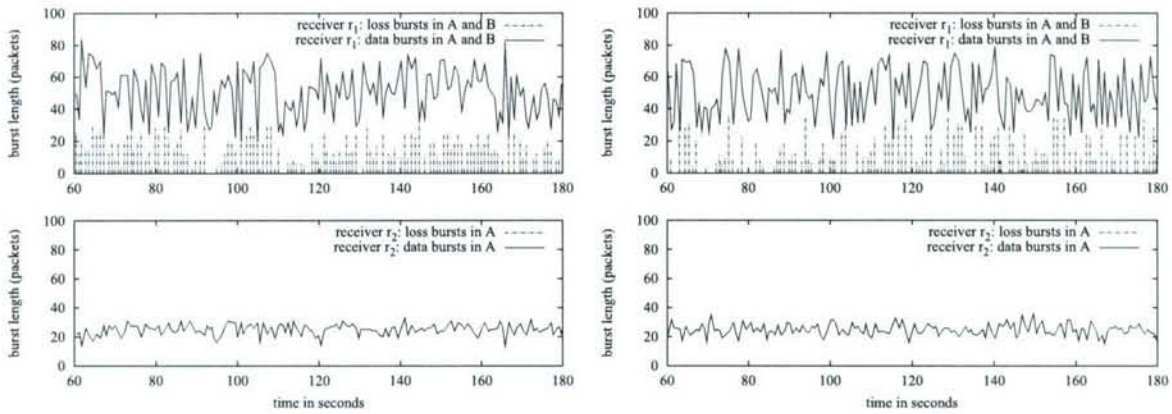
Where does Loss occur in a Datacenter? Datacenter networks have flat routing structures with no more than two or three hops on any end-to-end path. They are typically over-provisioned and of high quality, and packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are inexpensive and easily overloaded; even with high-capacity network interfaces, the commodity OS often drops packets due to buffer overflows caused by traffic spikes or high-priority threads occupying the CPU. Hence, our loss model is one of short packet bursts dropped at the end-host receivers at varying loss rates.

Figure 1 strongly indicates that loss in a datacenter is (a) bursty and (b) independent across end-hosts. In this experiment, a receiver r_1 joins two multicast groups A and B, and another receiver r_2 in the same switching segment joins only group A. From a sender located multiple switches away on the network, we send per-second data bursts of around 25 1KB packets to group A and simultaneously send a burst of 0-50 packets to group B, and measure packet loss at both receivers. We ran this experiment on two networks: a 64-node cluster at Cornell with 1.3 Ghz receivers and the Emulab testbed at Utah with 2 Ghz receivers, all nodes running Linux 2.6.12.

The top graphs in Figure 1 show the traffic bursts and loss bursts at receiver r_1, and the bottom graphs show the same information for r_2. We can see that r_1 gets overloaded and drops packets in bursts of size 1-30 packets, whereas r_2 does not drop any packets — importantly, around 30% of the packets dropped by r_1 are in group A, which is common to both receivers. Hence, loss is both bursty and independent across nodes. Together, these graphs indicate strongly that loss occurs due to buffer overflows at receiver r_1.

The example in Figure 1 is simplistic - each incoming burst of traffic arrives at the receiver within a small number of milliseconds - but conveys a powerful message: it is very easy to trigger significant bursty loss at datacenter end-hosts. The receivers in these experiments were running empty and draining packets continuously out of the kernel, with zero contention for the CPU or the network, whereas the settings of interest to us involve time-critical, possibly CPU-intensive applications running on top of the communication stack.

Further, we expect multi-group settings to intrinsically exhibit bursty incoming traffic of the kind emulated in this experiment — each node in the system receives data from multiple senders in multiple groups and it is likely that the inter-arrival time of data packets at a node will vary widely, even if the traffic rate at one sender or group is steady. In some cases, burstiness of traffic could also occur due to time-critical application behavior - for example, imagine an update in the value of a stock quote triggering off activity in several system components, which then multicast information to a replicated central data-



(a) Cornell 64-node Cluster

(b) Utah Emulab Testbed

Figure 1: Datacenter Loss is bursty and uncorrelated across nodes: receiver r_1 (top) joins groups A and B and exhibits bursty loss, whereas receiver r_2 (bottom) joins only group A and experiences zero loss.

store. If we assume that each time-critical component processes the update within a few hundred microseconds, and that inter-node socket-to-socket latency is around fifty microseconds (an actual number from our experimental cluster), the central datastore could easily see a sub-millisecond burst of traffic. In this case, the componentized structure of the application resulted in bursty traffic; in other scenarios, the application domain could be intrinsically prone to bursty input. For example, a financial calculator tracking a set of hundred equities with correlated movements might expect to receive a burst of a hundred packets in multiple groups almost instantaneously.

3 The Design of a Time-Critical Multicast Primitive

In recent years, multicast research has focused almost exclusively on application-level routing mechanisms, or overlay networks ([13] is one example), designed to operate in the wide-area without any existing router support. The need for overlay multicast stems from the lack of IP Multicast coverage in the modern internet, which in turn reflects concerns of administration complexity, scalability, and the risk of multicast 'storms' caused by misbehaving nodes. However, the homogeneity and comparatively limited size of datacenter networks pose few scalability and administration challenges to IP Multicast, making it a viable and attractive option in such settings. In this paper, we restrict ourselves to a more traditional definition of 'reliable multicast', as a reliability layer over IP Multicast. Given that the selection of datacenter hardware is typically influenced by commercial constraints, we believe that any viable solution for this context must be able to run on any mix of existing commodity routers and OS software; hence, we focus exclusively on mechanisms that act at the application-level, ruling out schemes which require router modification, such as PGM [19].

3.1 The Timeliness of (Scalable) Reliable Multicast Protocols

Reliable multicast protocols typically consist of three logical phases: *transmission* of the packet, *discovery* of packet loss, and *recovery* from it. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols, the time taken to discover packet loss dominates recovery latency heavily in the kind of settings we are interested in. The key insight is that *the discovery latency of reliable multicast protocols is usually inversely dependent on data rate*: for existing protocols, the rate of outgoing data at a single sender in a single group. Existing schemes for reliability in multicast can be roughly divided into the following categories:

ACK/timeout: RMTP [21], RMTP-II [22]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [21]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger time-outs at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

Gossip-Based: Bimodal Multicast [10], lpbcast [17]. Receivers periodically gossip histories of received packets

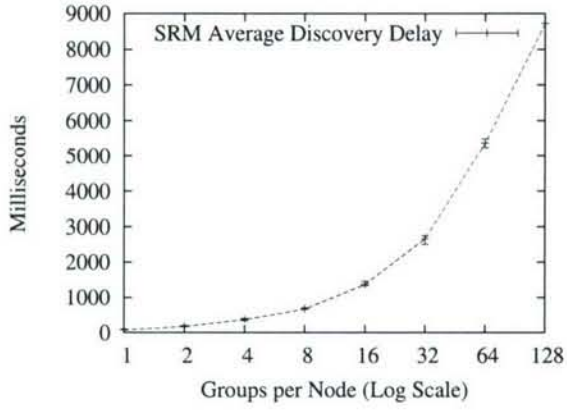


Figure 2: SRM's Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs.

with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery latency is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [10]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact.

NAK/Sender-based Sequencing: SRM [18]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery latency is thus proportional to the inter-send time at any single sender to a single group - a receiver can't discover a loss in a group until it receives the next packet from the same sender to that group - and consequently depends on the sender's data transmission rate to the group. To illustrate this point, we measured the performance of SRM as we increased the number of groups each node belonged in, keeping the throughput in the system constant by reducing the data rate within each group - as Figure 2 shows, discovery latency of lost packets degrades linearly as each node's bandwidth is increasingly fragmented and each group's rate goes down, increasing the time between two consecutive sends by a sender to the same group. Once discovery occurs in SRM, lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

Sender-based FEC [20, 23]: Forward Error Correction schemes involve multicasting redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating c repair packets for every r data packets, such that any r of the combined set of $r + c$ data and repair packets is sufficient to recover the original r data packets; we term this (r, c) parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out r data packets to that group. Fast, efficient encodings such as Tornado codes [11] make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, software distribution or internet radio.

Receiver-based FEC [9]: Of the above schemes, ACK-based protocols are intrinsically unsuited for time-critical multi-sender settings, while sender-based sequencing and FEC limit discovery latency to inter-send time at a single sender within a single group. Ideally, we would like discovery latency to be independent of inter-send time, and combine the scalability of a gossip-based scheme with the tunability of FEC. Receiver-based FEC, first introduced in the Slingshot protocol [9], provides such a combination: receivers generate FEC packets from incoming data and exchange these with other randomly chosen receivers. Since FEC packets are generated from *incoming* data at a receiver, the timeliness of loss recovery depends on the rate of data multicast in *the entire group*, rather than the rate at any given sender, allowing scalability in the number of senders to the group.

Slingshot is aimed at single-group settings, recovering from packet loss in time proportional to that group's data rate. Our goal with Ricochet is to achieve recovery latency dependent on the rate of data incoming at a node *across all groups*. Essentially, we want recovery of packets to occur as quickly in a thousand 10 Kbps groups as in a single 10 Mbps group, allowing applications to divide node bandwidth among thousands of multicast groups while maintaining time-critical packet recovery. To achieve this, we introduce Lateral Error Correction, a new form of receiver-generated FEC that probabilistically combines receiver-generated repair traffic across multiple groups to drive down packet recovery latencies.

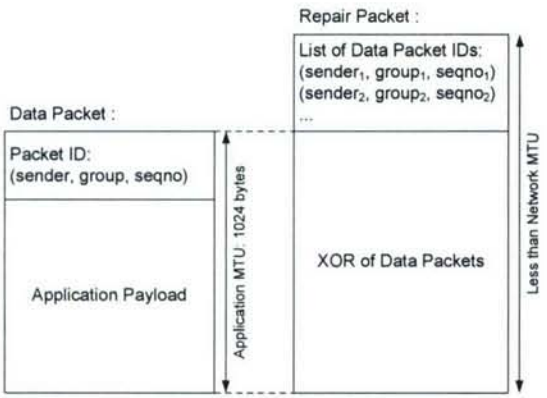


Figure 3: Ricochet Packet Structure

4 Lateral Error Correction and the Ricochet protocol

In Ricochet, each node belongs to a number of groups, and receives data multicast within any of them. The basic operation of the protocol involves generating XORs from incoming data and exchanging them with other randomly selected nodes. Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 3 shows the structure of these two packet types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from r data packets allows recovery of one of them, if all the other $r-1$ data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets.

At the core of Ricochet is the LEC engine running at each node that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind LEC is the notion that repair packets sent by a node to another node can be composed from data in any of the multicast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur within time that's inversely proportional to the aggregate rate of data in all these groups. Figure 4 illustrates this idea: n_1 has groups A and B in common with n_2, and hence it can generate and dispatch repair packets that contain data from both these groups. n_1 needs to wait only until it receives 5 data packets in either A *or* B before it sends a repair packet, allowing faster recovery of lost packets at n_2.

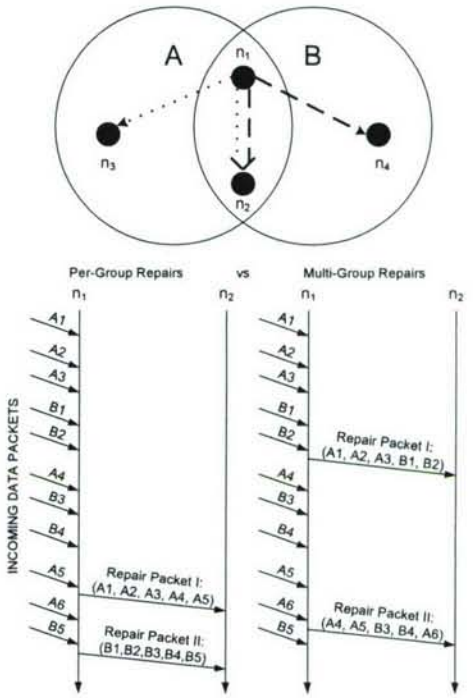


Figure 4: LEC in 2 Groups: Receiver n_1 can send repairs to n_2 that combine data from both groups A and B.

While combining data from different groups in outgoing repair packets drives down recovery time, it tampers with the coherent tunability that single group receiver-based FEC provides. The *rate-of-fire* parameter in receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every r data packets, c repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for LEC is to combine repair traffic for multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. Coherent, Tunable Per-Group Overhead: For every data packet that a node receives in a group with rate-of-fire (r, c), it sends out *an average of c repair packets* including that data packet to other nodes in the group.
2. Randomness: Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

LEC supports overlapping groups with the same r component and different c values in their rate-of-fire parameter. In LEC, the rate-of-fire parameter is translated into the following guarantee: For every data packet d that a node receives in a group with rate-of-fire (r, c), it selects an average of c nodes from the group randomly and sends each

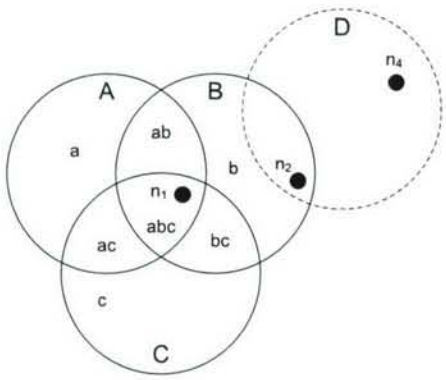


Figure 5: n_1 belongs to groups A, B, C: it divides them into disjoint regions abc, ab, ac, bc, a, b, c

of these nodes exactly one repair packet that includes d. In other words, the node sends an average of c repair packets containing d to the group. In the following section, we describe the algorithm that LEC uses to compose and dispatch repair packets while maintaining this guarantee.

4.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same LEC algorithm and supporting code runs at every node - and hence, we can describe its operation from the vantage point of a single node, n_1.

4.1.1 Regions

The LEC engine running at n_1 divides n_1's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that n_1 belongs to. Figure 5 shows the regions in a hypothetical system, where n_1 is in three groups, A, B and C. We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e, abc is a region formed by the intersection of A, B and C. In our example, the neighborhood set of n_1 is carved into seven regions: abc, ac, ab, bc, a, b and c, essentially the power set of the set of groups involved. Readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case; we are only concerned with non-empty intersections, the cardinality of which is bounded by the number of nodes in the system, as each node belongs to exactly one intersection (see Section 4.1.4). Note that n_1 does not belong to group D and is oblivious to it; it observes n_2 as belonging to region b, rather than bd, and is not aware of n_4's existence.

4.1.2 Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, LEC selects targets randomly from *each region*. The number of targets selected from a region is set such that:

1. It is proportional to the size of the region
2. The total number of targets selected, across regions, is equal to the c value of the group

Hence, for a given group A with rate-of-fire (r, c_A), the number of targets selected by LEC in a particular region, say abc, is equal to $c_A * \frac{|abc|}{|A|}$, where $|x|$ is the number of nodes in the region or group x. We denote the number of targets selected by LEC in region abc for packets in group A as c_A^{abc}. Figure 6 shows n_1 selecting targets for repairs from the regions of A.

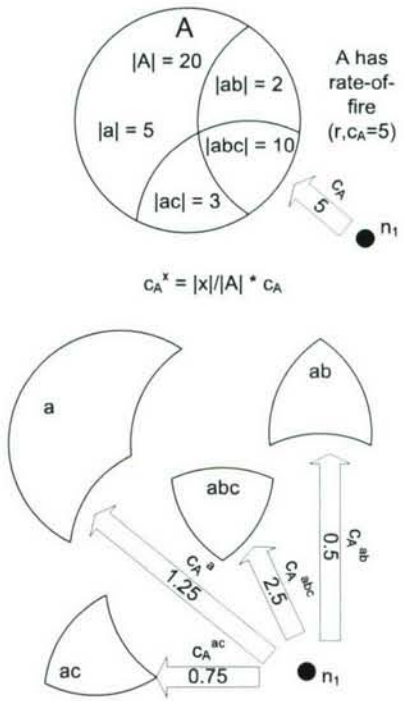


Figure 6: n_1 selects proportionally sized chunks of c_A from the regions of A

Note that LEC may pick a different number of targets from a region for packets in a different group; for example, c_A^{abc} differs from c_B^{abc}. Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size c has to be selected from the group, selecting targets from regions corresponds to *stratified sampling* [14], a technique from statistical theory.

4.1.3 Why select targets from regions?

Selecting targets from regions instead of groups allows LEC to construct repair packets from multiple groups; since we know that all nodes in region ab are interested in data from groups A and B, we can create composite

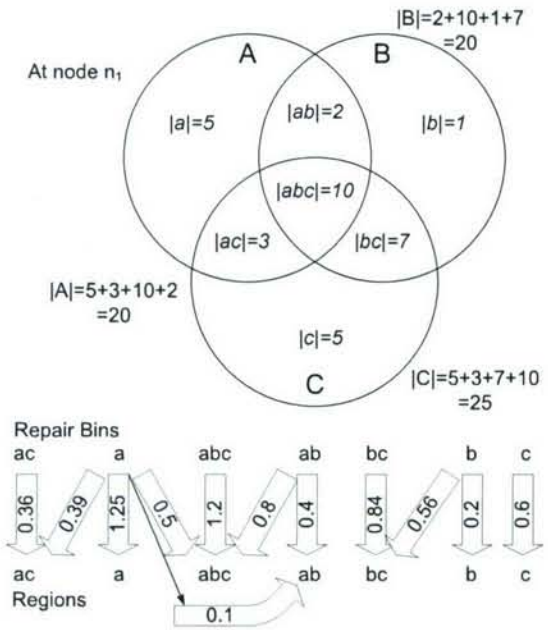


Figure 7: Mappings between repair bins and regions: the repair bin for ab selects 0.4 targets from region ab and 0.8 from abc for every repair packet. Here, $c_A = 5$, $c_B = 4$, and $c_C = 3$.

repair packets from incoming data packets in both groups and send them to nodes in that region.

Single-group receiver-based FEC [9] is implemented using a simple construct called a *repair bin*, which collects incoming data within the group. When a repair bin reaches a threshold size of r, a repair packet is generated from its contents and sent to c randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems simple; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size r, it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups A, B, and C, how many targets should we select from region abc for this repair packet - c_A^{abc}, c_B^{abc}, or c_C^{abc}? One possible solution is to pick the maximum of these values. If $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$, then we would select c_A^{abc}. However, a data packet in group B, when added to the repair bin for the region abc would be sent to an average of c_A^{abc} targets in the region; resulting in more repair packets containing that data packet sent to the region than required (c_B^{abc}), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group B than required by its (r, c_B) value; a similar argument holds for data packets in group C as well.

Algorithm 1 Algorithm for Setting Up Repair Bins

1: **Code at node n_i:**

2: **upon** Change in Group Membership **do**
3: **while** L not empty {*L is the list of regions*} **do**
4: Select and remove the region $R_i = abc...z$ from L with highest number of groups involved (break ties in any order)
5: Set $R_t = R_i$
6: **while** $R_t \neq \epsilon$ **do**
7: set c_{min} to $min(c_A^{R_t}, c_B^{R_t} ...)$, where {A,B,...} is the set of groups forming R_t
8: Set number of targets selected by R_i's repair bin from region R_t to c_{min}
9: Remove G from R_t, for all groups G where $c_G^{R_t} = c_{min}$
10: For each remaining group G' in R_t, set $c_{G'}^{R_t} = c_{G'}^{R_t} - c_{min}$

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups A and B than required, resulting in a lower fraction of packets recovered from LEC. To rectify this we send the additional compensating repair packets to the region abc from the repair bins for regions a and b. The repair bin for region a would select $c_A^{abc} - c_C^{abc}$ destinations, on an average, for every repair packet it generates; this is in addition to the c_A^a destinations it selects from region a.

A more sophisticated version of the above strategy involves iteratively obtaining the required repair packets from regions involving the remaining groups; for instance, we would have the repair bin for ab select the minimum of c_A^{abc} and c_B^{abc} - which happens to be c_B^{abc} - from abc, and then have the repair bin for a select the remainder value, $c_A^{abc} - c_B^{abc}$, from abc. Algorithm 1 illustrates the final approach adopted by LEC, and Figure 7 shows the output of this algorithm for an example scenario. A repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

4.1.4 Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that n_1 is part of. The algorithm has complexity $O(I \cdot d)$, where I is the number of populated regions (i.e, with one or more nodes in them), and d is the maximum number of groups that form a region. Note that I at n_1 is bounded from above by the cardinality of the set of nodes that share a multicast

group with n_1, since regions are disjoint and each node exists in exactly one of them. d is bounded by the number of groups that n_1 belongs to.

4.2 Implementation Details

Our implementation of Ricochet is in Java. Below, we discuss the details of the implementation, along with the performance optimizations involved - some obvious and others subtle.

4.2.1 Repair Bins

A Ricochet repair bin is a lightweight structure holding an XOR and a list of data packets, and supporting an *add* operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in Figure 7, the repair bin for abc has to select 1.2 targets from abc, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

4.2.2 Staggering for Bursty Loss

A crucial algorithmic optimization in Ricochet is *staggering* - also known as interleaving [23] - which provides resilience to bursty loss. Given a sequence of data packets to encode, a stagger of 2 would entail constructing one repair packet from the 1st, 3rd, 5th... packets, and another repair packet from the 2nd, 4th, 6th... packets. The stagger value defines the number of repairs simultaneously being constructed, as well as the distance in the sequence between two data packets included in the same repair packet. Consequently, a stagger of i allows us to tolerate a loss burst of size i while resulting in a proportional slowdown in recovery latency, since we now have to wait for $O(i*r)$ data packets before despatching repair packets.

In conventional sender-based FEC, staggering is not a very attractive option, providing tolerance to very small bursts at the cost of multiplying the already prohibitive loss discovery latency. However, LEC recovers packets so quickly that we can tolerate a slowdown of a factor of ten without leaving the tens of milliseconds range; additionally, a small stagger at the sender allows us to tolerate very large bursts of lost packets at the receiver, especially since the burst is dissipated among multiple groups and senders. Ricochet implements a stagger of i by the simple expedient of duplicating each logical repair bin into i instances; when a data packet is added to the logical repair bin, it is actually added to a particular instance of the repair bin, chosen in round-robin fashion. Instances of a duplicated repair bin behave exactly as single repair bins do, generating repair packets and sending them to regions when they get filled up.

4.2.3 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [24]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the LEC algorithm. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at n_1 - and consequently the group and intersection sizes - does not include n_1 itself.

4.2.4 Membership and Failure Detection

Ricochet can plug into any existing membership and failure detection infrastructure, as long as it is provided with reasonably up-to-date views of per-group membership by some external service. In our implementation, we use simple versions of Group Membership (GMS) and Failure Detection (FD) services, which execute on high-end server machines. If the GMS receives a notification from the FD that a node has failed, or it receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s). The GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node n_{55} joins group A, the update sent by the GMS to every node in A would be a 3-tuple: *(Join, A, n_{55})*. Individual nodes process these updates to construct multi-group views relative to their own membership.

Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this can be easily done by partitioning the service on group id and running each partition on a different server. For instance, one machine is responsible for groups A and B, another for C and D, and so on. Similarly, the FD can be partitioned on a topological criterion; one machine on each rack is responsible for monitoring other nodes on the rack by pinging them periodically. For fault-tolerance, each partition of the GMS can be replicated on multiple machines using a strongly consistent protocol like Paxos. The FD can have a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain. We believe that

such a semi-centralized solution is appropriate and sufficient in a datacenter setting, where connectivity and membership are typically stable. Crucially, the protocol itself does not need consistent membership, and degrades gracefully with the degree of inconsistency in the views; if a failed node is included in a view, performance will dip fractionally in all the groups it belongs to as the repairs sent to it by other nodes are wasted.

4.2.5 Performance

Since Ricochet creates LEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to: Instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to c (from the rate-of-fire) in the worst case - which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

4.2.6 Buffering and Loss Control

LEC - like any other form of FEC - works best when losses are not in concentrated bursts. Ricochet maintains an application-level buffer with the aim of minimizing in-kernel losses, serviced by a separate thread that continuously drains packets from the kernel. If memory at end-hosts is constrained and the application-level buffer is bounded, we use customized packet-drop policies to handle overflows: a randomly selected packet from the buffer is dropped and the new packet is accommodated instead. In practice, this results in a sequence of almost random losses from the buffer, which are easy to recover using FEC traffic. Whether the application-level buffer is bounded or not, it ensures that packet losses in the kernel are reduced to short bursts that occur only during periods of overload or CPU contention. We evaluate Ricochet against loss bursts of up to 100 packets, though in practice we expect the kind of loss pattern shown in 1, where few bursts are greater than 20-30 packets, even with highly concentrated traffic spikes.

4.2.7 NAK Layer for 100% Recovery

Ricochet recovers a high percentage of lost packets via the proactive LEC traffic; for certain applications, this probabilistic guarantee of packet recovery is sufficient and even desirable in cases where data 'expires' and there is no utility in recovering it after a certain number of milliseconds. However, the majority of applications require 100% recovery of lost data, and Ricochet uses a reactive NAK layer to provide this guarantee. If a receiver does not recover a packet through LEC traffic within a timeout period after discovery of loss, it sends an explicit NAK to the sender and requests a retransmission. While this NAK layer does result in extra reactive repair traffic, two factors separate it from traditional NAK mechanisms: firstly, recovery can potentially occur very quickly - within a few hundred milliseconds - since for almost all lost packets discovery of loss takes place within milliseconds through LEC traffic. Secondly, the NAK layer is meant solely as a backup mechanism for LEC and responsible for recovering a very small percentage of total loss, and hence the extra overhead is minimal.

4.2.8 Optimizations

Ricochet maintains a buffer of unusable repair packets that enable it to utilize incoming repair packets better. If one repair packet is missing exactly one more data packet than another repair packet, and both are missing at least one data packet, Ricochet obtains the extra data packet by XORing the two repair packets. Also, it maintains a list of unusable repair packets which is checked intermittently to see if recent data packet recoveries and receives have made any old repair packets usable.

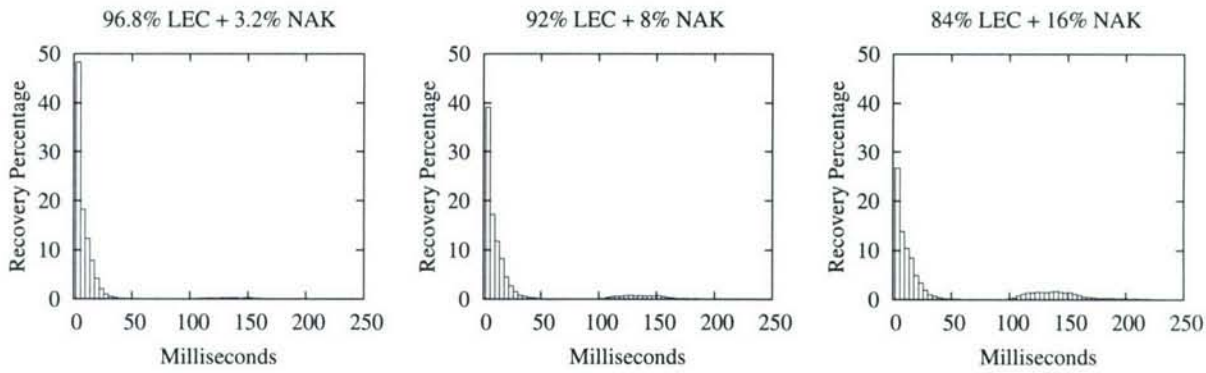
4.2.9 Message Ordering

As presented, Ricochet provides multicast reliability but does not deliver messages in the same order at all receivers. We are primarily concerned with building an extremely rapid multicast primitive that can be used by applications that require unordered reliable delivery as well as layered under ordering protocols with stronger delivery properties. For instance, Ricochet can be used as a reliable transport by any of the existing mechanisms for total ordering [16] — in separate work [8], we describe one such technique that predicts out-of-order delivery in datacenters to optimize ordering delays.

5 Evaluation

We evaluated our Java implementation of Ricochet on a 64-node cluster, comprising of four racks of 16 nodes each, interconnected via two levels of switches. Each node has a single 1.3 GHz CPU with 512 Mb RAM, runs Linux 2.6.12 and has two 100 Mbps network interfaces, one for control and the other for experimental traffic. Typical socket-to-socket latency within the cluster is around 50 microseconds. In the following experiments, for a given loss rate L, three different loss models are used:

· **uniform** - also known as the Bernoulli model [25] - refers to dropping packets with uniform probability equal to the loss rate L.



(a) 10% Loss Rate **(b) 15% Loss Rate** **(c) 20% Loss Rate**

Figure 8: Distribution of Recoveries: LEC + NAK for varying degrees of loss

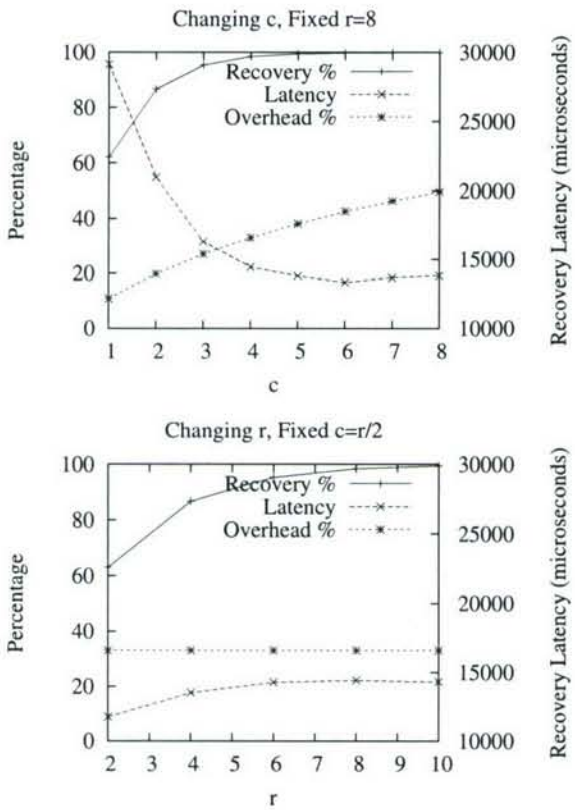


Figure 9: Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire (r, c).

· **bursty** involves dropping packets in equal bursts of length b. The probability of starting a loss burst is set so that each burst is of exactly b packets and the loss rate is maintained at L. This is not a realistic model but allows us to precisely measure performance relative to specific burst lengths.

· **markov** drops packets using a simple 2-state markov chain, where each node alternates between a lossy and a lossless state, and the probabilities are set so that the average length of a loss burst is m and the loss rate is L, as described in [25].

In experiments with multiple groups, nodes are assigned to groups at random, and the following formula is used to relate the variables in the grouping pattern: $n * d = g * s$, where n is the number of nodes in the system (64 in most of the experiments), d is the degree of membership, i.e. the number of groups each node joins, g is the total number of groups in the system, and s is the average size of each group. For example, in a 16-node setting where each node joins 512 groups and each group is of size 8, g is set to $\frac{16*512}{8} \approx 1024$. Each node is then assigned to 512 randomly picked groups out of 1024. Hence, the grouping patterns for each experiment is completely represented by a (n, d, s) tuple.

For every run, we set the sending rate at a node such that the total system rate of incoming messages is 64000 packets per second, or 1000 packets per node per second. Data packets are 1K bytes in size. Each point in the following graphs - other than Figure 8, which shows distributions for single runs - is an average of 5 runs. A run lasts 30 seconds and produces $\approx$ 2 million receive events in the system.

5.1 Distribution of Recoveries in Ricochet

First, we provide a snapshot of what typical packet recovery timelines look like in Ricochet. Earlier, we made

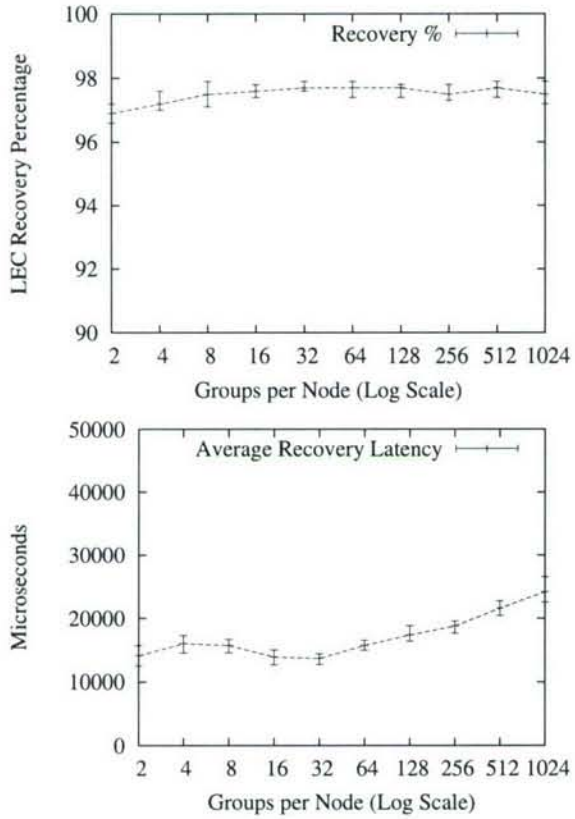


Figure 10: Scalability in Groups

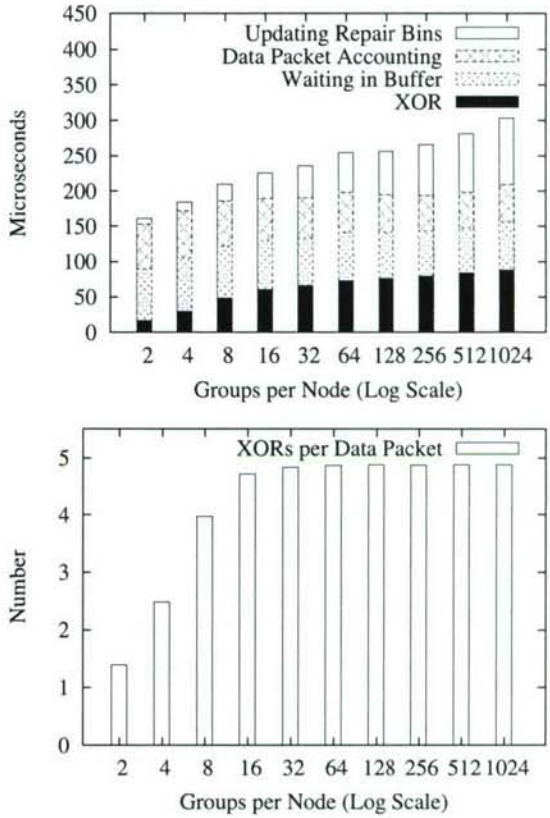
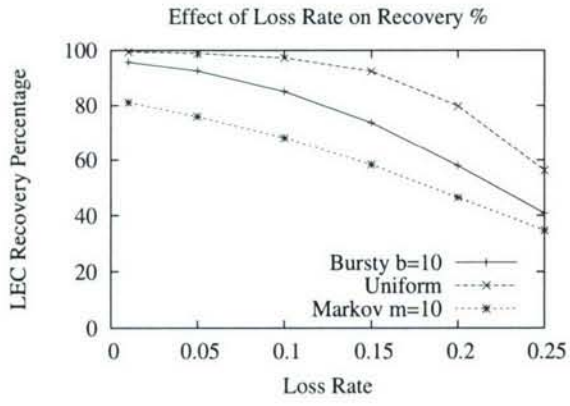


Figure 11: CPU time and XORs per data packet

the assertion that Ricochet discovers the loss of almost all packets very quickly through LEC traffic, recovers a majority of these instantly and recovers the remainder using an optional NAK layer. In Figure 8, we show the histogram of packet recovery latencies for a 16-node run with degree of membership $d = 128$ and group size $s = 10$. We use a simplistic NAK layer that starts unicasting NAKs to the original sender of the multicast 100 milliseconds after discovery of loss, and retries at 50 millisecond intervals. Figure 8 shows three scenarios: under uniform loss rates of 10%, 15%, and 20%, different fractions of packet loss are recovered through LEC and the remainder via reactive NAKs. These graphs illustrate the meaning of the LEC recovery percentage: if this number is high, more packets are recovered very quickly without extra traffic in the initial segment of the graphs, and less reactive overhead is induced by the NAK layer. Importantly, even with a recovery percentage as low as 84% in Figure 8(c), we are able to recover all packets within 250 milliseconds with a crude NAK layer due to early LEC-based discovery of loss. For the remaining experiments, we will switch the NAK layer off and focus solely on LEC performance; also, since we found this distribution of recovery latencies to be fairly representative, we present only the percentage of lost packets recovered using LEC and the average latency of these recoveries. Experiment Setup: $(n = 16, d = 128, s = 10)$, Loss Model: Uniform, [10%, 15%, 20%].

5.2 Tunability of LEC in multiple groups

The Slingshot protocol [9] illustrated the tunability of receiver-generated FEC for a single group; we include a similar graph for Ricochet in Figure 9, showing that the rate-of-fire parameter (r, c) provides a knob to tune LEC's recovery characteristics. In Figure 9.a, we can see that increasing the c value for constant $r = 8$ increases the recovery percentage and lowers recovery latency by expending more overhead - measured as the percentage of repair packets to all packets. In Figure 9.b, we see the impact of increasing r, keeping the ratio of c to r - and consequently, the overhead - constant. For the rest of the experiments, we set the rate-of-fire at $(r = 8, c = 5)$. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Uniform, 1%.



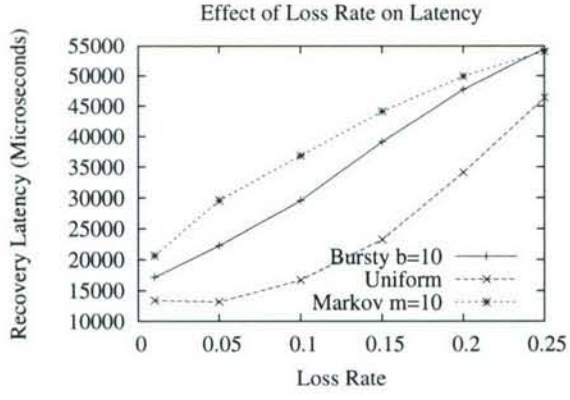
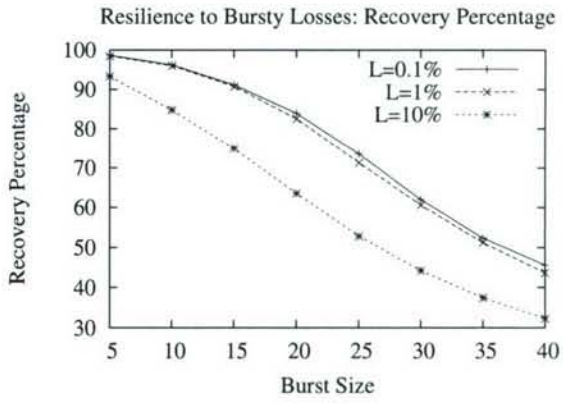


Figure 12: Impact of Loss Rate on LEC



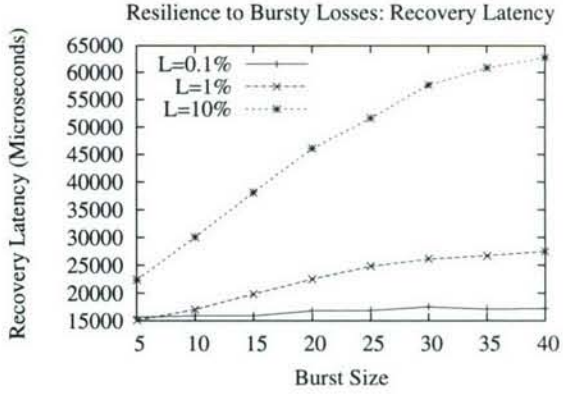


Figure 13: Resilience to Burstiness

5.3 Scalability

Next, we examine the scalability of Ricochet to large numbers of groups. Figure 10 shows that increasing the degree of membership for each node from 2 to 1024 has almost no effect on the percentage of packets recovered via LEC, and causes a slow increase in average recovery latency. The x-axis in these graphs is log-scale, and hence a straight line increase is actually logarithmic with respect to the number of groups and represents excellent scalability. The increase in recovery latency towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section.

For a comparison point, we refer readers back to SRM's discovery latency in Figure 2: in 128 groups, SRM discovery took place at 9 seconds. In our experiments, SRM recovery took place roughly 4 seconds after discovery in all cases. While fine-tuning the SRM implementation for clustered settings should eliminate that 4 second gap between discovery and recovery, at 128 groups Ricochet surpasses SRM's best possible recovery performance of 5 seconds by between 2 and 3 orders of magnitude.

Though Ricochet's recovery characteristics scale well in the number of groups, it is important that the computational overhead imposed by the protocol on nodes stays manageable, given that time-critical applications are expected to run over it. Figure 11 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a log x-axis shows that per-packet processing time increases logarithmically with the number of groups - doubling the number of groups results in a constant increase in processing time. The increase in processing time towards the latter half of the graph is due to the increase in the number of repair bins with the number of groups. While we considered 1024 groups adequate scalability, Ricochet can potentially scale to more groups with further optimization, such as creating bins only for occupied regions. In the current implementation, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second. Figure 11 also shows the average number of XORs per incoming data packet. As stated in section 4.2.2, the number of XORs stays under 5 - the value of c from the rate-of-fire (r, c). Experiment Setup: $(n = 64, d = *, s = 10)$, Loss Model: Uniform, 1%.

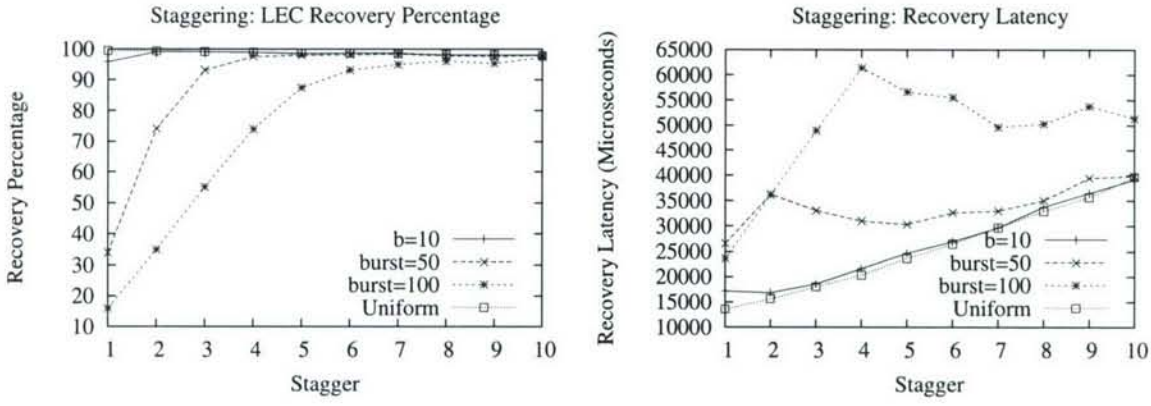


Figure 14: Staggering allows Ricochet to recover from long bursts of loss.

5.4 Loss Rate and LEC Effectiveness

Figure 12 shows the impact of the Loss Rate on LEC recovery characteristics, under the three loss models. Both LEC recovery percentages and latencies degrade gracefully: with an unrealistically high loss rate of 25%, Ricochet still recovers 40% of lost packets at an average of 60 milliseconds. For uniform and bursty loss models, recovery percentage stays above 90% with a 5% loss rate; markov does not fare as well, even at 1% loss rate, primarily because it induces bursts much longer than its average of 10 - the max burst in this setting averages at 50 packets. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: *.

5.5 Resilience to Bursty Losses

As we noted before, a major criticism of FEC schemes is their fragility in the face of bursty packet loss. Figure 13 shows that Ricochet is naturally resilient to small loss bursts, without the stagger optimization - however, as the burst size increases, the percentage of packets recovered using LEC degrades substantially. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty.

However, switching on the stagger optimization described in Section 4.2.2 increases Ricochet's resilience to burstiness tremendously, without impacting recovery latency much. Figure 14 shows that setting an appropriate stagger value allows Ricochet to handle large bursts of loss: for a burst size as large as 100, a stagger of 6 enables recovery of more than 90% lost packets at an average latency of around 50 milliseconds. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty, 1%.

5.6 Effect of Group and System Size

What happens to LEC performance when the average group size in the cluster is large compared to the total number of nodes? Figure 15 shows that recovery percentages are almost unaffected, staying above 99% in this scenario, but recovery latency is impacted by more than a factor of 2 as we triple group size from 16 to 48 in a 64-node setting. Note that this measures the impact of the size of the group relative to the entire system; receiver-based FEC has been shown to scale well in a single isolated group to hundreds of nodes [9]. Experiment Setup: $(n = 64, d = 128, s = *)$, Loss Model: Uniform, 1%.

While we could not evaluate to system sizes beyond 64 nodes, Ricochet should be oblivious to the size of the entire system, since each node is only concerned with the groups it belongs to. We ran 4 instances of Ricochet on each node to obtain an emulated 256 node system with each instance in 128 groups, and the resulting recovery percentage of 98% - albeit with a degraded average recovery latency of nearly 200 milliseconds due to network and CPU contention - confirmed our intuition of the protocol's fundamental insensitivity to system size.

6 Future Work

One avenue of research involves embedding more complex error codes such as Tornado [11] in LEC; however, the use of XOR has significant implications for the design of the algorithm, and using a different encoding might require significant changes. LEC uses XOR for its simplicity and speed, and as our evaluation showed, we obtain properties on par with more sophisticated encodings, including tunability and burst resilience. We plan on replacing our simplistic NAK layer with a version optimized for bulk transfer, providing an efficient backup for LEC when sustained bursts occur of hundreds of packets or more. Another line of work involves making the parameters for LEC - such as rate-of-fire and stagger - adaptive, reacting to meet varying load and network characteristics. We are currently working with industry partners to layer Ricochet under data distribution, publish-subscribe and web-service interfaces, as well as building protocols with stronger ordering and atomicity properties over it.

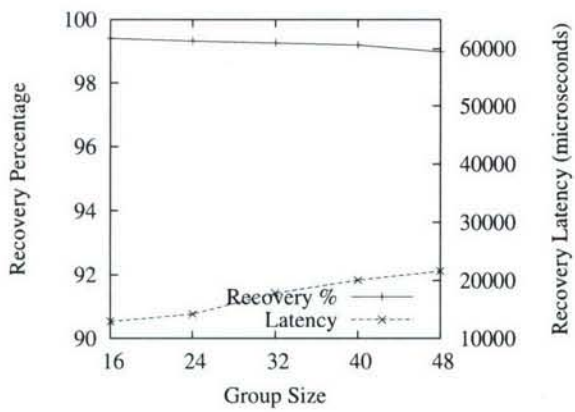


Figure 15: Effect of Group Size

7 Conclusion

We believe that the next generation of time-critical applications will execute on commodity clusters, using the techniques of massive redundancy, fault-tolerance and scalable communication currently available to distributed systems practitioners. Such applications will require a multicast primitive that delivers data at the speed of hardware multicast in failure-free operation and recovers from packet loss within milliseconds irrespective of the pattern of usage. Ricochet provides applications with massive scalability in multiple dimensions - crucially, it scales in the number of groups in the system, performing well under arbitrary grouping patterns and overlaps. A clustered communication primitive with good timing properties can ultimately be of use to applications in diverse domains not normally considered time-critical - e-tailers, online webservers and enterprise applications, to name a few.

Acknowledgments

We received invaluable comments from Dave Andersen, Danny Dolev, Tudor Marian, Art Munson, Robbert van Renesse, Emin Gun Sirer, Niraj Tolia and Einar Vollset. We would like to thank our shepherd Mike Dahlin, as well as all the anonymous reviewers of the paper.

References

[1] Bea weblogic. `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic`, 2006.

[2] Gemstone gemfire. `http://www.gemstone.com/products/gemfire/enterprise.php`, 2006.

[3] Ibm websphere. `www.ibm.com/software/webservers/appserv/was/`, 2006.

[4] Jboss. `http://labs.jboss.com/portal/`, 2006.

[5] Real-time innovations data distribution service. `http://www.rti.com/products/data_distribution/index.html`, 2006.

[6] Tangosol coherence. `http://www.tangosol.com/html/coherence-overview.shtml`, 2006.

[7] Tibco rendezvous. `http://www.tibco.com/software/messaging/rendezvous.jsp`, 2006.

[8] M. Balakrishnan, K. Birman, and A. Phanishayee. Plato: Predictive latency-aware total ordering. In *IEEE SRDS*, 2006.

[9] M. Balakrishnan, S. Pleisch, and K. Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*, 2005.

[10] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98 Conference*, pages 56–67, New York, NY, USA, 1998. ACM Press.

[12] Y. Chawathe, S. McCanne, and E. A. Brewer. Rmx: Reliable multicast for heterogeneous networks. In *INFOCOM*, pages 795–804, 2000.

[13] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *ACM SIGCOMM*, pages 55–67, New York, NY, USA, 2001. ACM Press.

[14] W. G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.

[15] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[17] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[18] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[19] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The pgm reliable multicast protocol. *IEEE Network*, 17(1):16–22, Jan 2003.

[20] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[21] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[22] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, Apr. 1998. IETF Internet Draft.

[23] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.

[24] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[25] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *INFOCOM*, pages 345–352, 1999.

MISTRAL: Efficient Flooding in Mobile Ad-hoc Networks*

Stefan Pleisch† Mahesh Balakrishnan‡ Ken Birman‡ Robbert van Renesse‡

†Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland

‡Department of Computer Science
Cornell University, Ithaca, NY 14853, USA

stefan.pleisch@epfl.ch {mahesh|ken|rvr}@cs.cornell.edu

ABSTRACT

Flooding is an important communication primitive in mobile ad-hoc networks and also serves as a building block for more complex protocols such as routing protocols. In this paper, we propose a novel approach to flooding, which relies on proactive compensation packets periodically broadcast by every node. The compensation packets are constructed from dropped data packets, based on techniques borrowed from forward error correction. Since our approach does not rely on proactive neighbor discovery and network overlays it is resilient to mobility.

We evaluate the implementation of Mistral through simulation and compare its performance and overhead to purely probabilistic flooding. Our results show that Mistral achieves a significantly higher node coverage with comparable overhead.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*;
C.4 [**Computer-Communication Networks**]: Performance of Systems—*Reliability, availability, and serviceability*;
C.4 [**Computer-Communication Networks**]: Performance of Systems—*Fault tolerance*

General Terms

Algorithms, reliability

Keywords

Mobile ad hoc networks, MANET, flooding, forward error correction, compensation

*Our effort is supported by the Swiss National Science Foundation (SNF), NSF Trust STC, the NSP NetNOSS program, and the DARPA ACERT program.

MobiHoc'06, May 22–25, 2006, Florence, Italy.

1. INTRODUCTION

Mobile ad hoc networks (MANETs) have received much attention in recent years. A MANET is a multi-hop wireless network without fixed infrastructure, in which nodes can be mobile. MANETs are increasingly important because wireless communication is rapidly becoming ubiquitous. Potential applications range from military and disaster response applications to more traditional urban problems such as finding desired products or services in a city. The devices themselves are diverse, including PDAs, cell phones, sensors, laptops, etc. Many new protocols have been proposed to solve the technical problems confronted in MANETs and to offer platform support for applications that collect and exploit the data available in such settings.

Because of the lack of a fixed communication infrastructure, flooding in MANETs [10] is an important communication primitive and also serves as a building block for more complex protocols such as AODV [21] or ODMRP [16]. *Flooding* is the mechanism by which a node, receiving flooded message m for the first time, rebroadcasts m once. We distinguish between flooding and *broadcast*, which is a transmission that is received by all nodes within transmission range of the broadcasting node. Flooding usually covers all the nodes in a network, but can also be limitted to a set of nodes that is defined by a geographical area (also called *geocast flooding* [14]) or by the time-to-live (TTL) parameter of m. Thus, a node receiving the flooded message only rebroadcasts it if it is within the specified area or if the message's TTL is greater than 0.

Unfortunately, flooding has been shown to be susceptible to contention even in reasonably dense networks [18]. Indeed, flooding leads to a large amount of redundant messages that consume scarce resources such as bandwidth and power and cause contention, collisions and thus additional packet loss. Every node receives the message from every neighbor within transmission range, except when messages are lost due to contention and collisions. This problem is known as the *broadcast storm* problem [18]. Because flooding is important in MANET applications, there is a clear need for storm-resistant flooding protocols that operate efficiently. However, reducing the number of redundant broadcasts leads to a lower degree of reliability. Hence, the challenge we face is to strike a balance between message overhead (i.e., the level of redundancy) and reliability.

To reduce the number of redundant messages, two ba-

sic classes of mechanisms have been proposed: (1) imposing a (partial) routing overlay structure; and (2) selectively dropping messages. Approaches in (1) build and maintain a (partial) routing overlay structure in the ad hoc network, which is used to efficiently broadcast the flooded message. For instance, only nodes that are part of a multicast tree rebroadcast the message [20]. Other approaches in this category are [3, 8, 17]. With mobile nodes the underlying routing structure needs to be frequently changed, incurring high maintenance costs and generally reduced reliability during the restructuring. In contrast, approaches in (2) do not rely on an explicit underlying routing structure. Instead, each node uses local information to make an independent decision whether to rebroadcast or to drop the flooded message. The simplest approach in this class is purely probabilistic flooding [18], in which messages are rebroadcast with a certain fixed probability. While probabilistic flooding reduces the number of broadcasts, when applied naively it simply recreates our earlier problem: poorly connected nodes (those with few neighbors) may fail to receive a flooded message. This consideration has motivated a number of more complex approaches, such as the algorithms given in [28, 24].

In our paper, we focus on class (2) but propose a new mechanism to reduce the number of missed flooded messages. We start with purely probabilistic flooding [18] but compensate for dropped data packets by periodically broadcasting *compensation packets*. Every compensation packet encodes a set of packets that have been dropped (i.e., that are not rebroadcast) by the sender. A node's neighbors, upon receipt of such a packet, can recover missing packets if it already has received and buffered a sufficient percentage of the packets that were used in constructing the compensation packet.

Even when a node has lost too many packets to reconstruct missing data, the compensation packets provide information that can be used to identify the loss. We include a secondary recovery mechanism that kicks in when a node discovers an unrecoverable loss, and part of our task in the evaluation presented here is to quantify the tradeoff between the additional message overhead versus increased reliability.

We have implemented Mistral and simulate its performance on JiST/SWANS, a simulation package that lets the developer run code in an emulated environment. Our results show that compensation packets significantly increase coverage when compared to probabilistic flooding with comparable overhead.

The remainder of the paper is structured as follows: Section 2 overviews the problem of flooding and places our work in the context of earlier work. In Section 3 we introduce the Mistral algorithm. Section 4 provides a simple analysis of Mistral. In Section 5, we present the simluation results and measure Mistral's performance. We conclude the paper with Section 6.

2. FLOODING IN MANETS

In any flooding mechanism, one must balance reliability against message overhead. On the one hand, increasing reliability generally involves sending a greater number of redundant messages and thus incurs a higher message overhead. In this worst case, the system risks provoking broadcast storms. Yet redundant messages are needed to reach all nodes and to recover from packet loss, hence reducing the overhead will generally decrease reliability.

The broadcast storm problem is so common in flooding algorithms that it has engendered a whole area of research. Storm-sensitive flooding approaches can be broadly classified into two classes: *local-knowledge-based* and *overlay-based*. Local-knowledge-based approaches decide on whether to rebroadcast or drop a flooded message solely on the basis of local information. Most commonly, they use information from received broadcasts to adaptively determine the forwarding policy. Such algorithms are a natural fit for MANETs, as they do not need to maintain any kind of complex node-to-node state that might need to be adapted in the event of mobility or other topology changes. In contrast, overlay-based approaches structure the node field according to some (local) topology, and then use topological information to efficiently implement flooding and reliability. The problem here is that if nodes have low quality connections to neighbors and/or are in motion, the overlay structure must be adapted. As a consequence, a high rate of management messages may be required, and if a flooded message is propagated while the overlay is out of date, that message may experience a high loss rate. In the worst case, the system might end up in a state of churn, constantly adapting the overlay but never managing to achieve the high quality of flooding that the overlay is intended to support.

We now briefly overview existing work and assign it to the corresponding class. For reasons of brevity, our review is deliberately partial; we focus on results that inspired our work here, or that have been widely cited in the literature. For a more comprehensive overview that includes a comparison of some of the major flooding approaches the reader is referred to [26].

2.1 Overlay-Based Approaches

As just indicated, we use the term *overlay* very broadly. For us, an overlay-based approaches is an algorithm that superimposes a routing structure onto the ad hoc network in support of flooding and rebroadcast. Depending on the position of a node in this overlay, it decides to either rebroadcast a flooded packet, or to only process and then drop it. While overlays provide a convenient mechanism to reduce the message overhead of flooding and to increase reliability, they suffer from the need to reconfigure the overlay when connectivity changes or if the nodes are mobile. Restructuring adds overhead but also increases the likelihood that messages will be lost, and thus may decrease coverage of the flooding protocol.

Ni *et al.* [18] propose to structure the nodes into clusters. Their solution rebroadcasts a packet in a manner that depends on the node's position in the cluster: only cluster head and gateway nodes rebroadcast.

In [8], the goal is to provide low-latency flooding. This is in part achieved by minimizing the collisions and interference. Gandi *et al.* show that an optimal solution to this problem is NP complete, instead, they propose an approximation algorithm. They construct a multicast tree and compute a rebroadcasting schedule such that the expected rate of collisions will be low.

Other approaches are based on the approximation of (minimal) connected dominating sets (MCDS), e.g., [5] [3]. Informally, a dominating set (DS) contains a subset of all nodes such that every node not in the DS is adjacent to one in the DS. Thus, a DS creates a virtual backbone that can be used to efficiently flood messages. It has been shown that

the creation of an MCDS is NP-complete. Thus, most approaches attempt to find a sufficiently good approximation to a MCDS.

A number of approaches rely on two-hop neighbor information to select nodes that rebroadcast the message. These approaches require that *hello* messages containing neighbor information are exchanged between the nodes.

For instance, in the Double-Covered Broadcast (DCB) [17], node n collects information about the two-hop neighbor set. Among its one-hop neighbors it then picks nodes that rebroadcast the message (called *forward node*) such that (1) the rebroadcast by the forward node covers the two-hop neighbors, and (2) the one-hop neighbors that are no forward nodes are within range of at least two rebroadcasts by forward nodes. The reception of the message by the forward node is implicitly acknowledged when n overhears the rebroadcast.

The scalable broadcast algorithm (SBA) [20] also uses two-hop neighbor knowledge, but employs a different approach to select the forward nodes.

With node mobility, the two-hop neighbor sets need to be updated frequently. Otherwise, the neighbor sets become outdated and reliability drops (as observed in [17]).

2.2 Local-Knowledge-Based Approaches

Local-knowledge-based approaches generally decide on a per-node basis whether to rebroadcast a particular flooded message. In the simplest case, each node flips a coin and rebroadcasts messages with a certain probability p [18]. We call this approach *purely probabilistic flooding (PPF)*.

There are a number of variants on this basic idea. For example, one set of algorithms base the rebroadcast decision either on the number of already overheard rebroadcasts, or on the distance or location of the overheard rebroadcast's sender [18]. The idea underlying these schemes is that the additional coverage gained by rebroadcasting decreases with the number of overheard rebroadcasts and decreasing distance to neighboring rebroadcasting nodes. However, it takes time to collect these statistics, delaying the rebroadcast decision, hence a potentially high latency is introduced to every flooded message. In [25], Tseng *et al.* extend earlier approaches in [18] to allow nodes to dynamically adapt threshold values such as the rebroadcast counter.

In [28] Zhang and Agrawal propose an approach that is a combination of the counter-based and probabilistic method of [18]. Instead of using a static rebroadcast probability p, they adjust p according to the information collected by the counters. While this makes p adaptable, it becomes dependent upon other fixed parameters that need to be carefully selected (e.g., timeouts).

Dynamic Gossip [24] relies on local density awareness to adjust the rebroadcast probability p of the one-hop neighbors. Its correctness and suitability relies on the assumption that the nodes are uniformly distributed. Density information is collected using a relay-ping method.

In [15], Kowalski and Pelc propose a broadcasting algorithm with optimal lower bounds in their model. They consider only stationary nodes and ajust the broadcast probability accordingly.

Haas *et al.* [9] study what they term a *phase transition phenomena*. This work shows that purely probabilistic flooding (called *gossiping* in [9]) in an ad hoc network has a *bimodal* delivery distribution. Their simulations reveal that either almost every node receives the message, or virtually none. To reduce the likelihood of the latter case, they explore a variety of approaches, such as adapting the rebroadcast probability to the density or the distance to the flooding source. Sasson *et al.* [23] theoretically explore the same phenomena based on percolation theory and conclude that there exists a threshold $\bar{p} < 1$ such that for any $p > \bar{p}$ the node coverage is close to 1, while for $p < \bar{p}$ the coverage is very low. Hence, increasing p much beyond $\bar{p}$ is not very useful.

Any approach that bases rebroadcast decision on observation of neighbors and on overheard broadcasts is at risk of using stale information if nodes might move before the information is used. MANETs, of course, can have a high degree of mobility, hence neither of these approaches is ideal.

Mistral's compensation mechanisms is orthogonal to these approaches. Indeed, were we building a production deployment of flooding in a real-world setting, we would be inclined to combine Mistral with one of these others (as should be clear, the ideal choice of underlying mechanism depends upon the anticipated density of nodes and level of mobility; no single solution stands out as uniformly superior to the others). By using such a hybrid scheme, we could parameterize the underlying solution to keep overheads low, accepting a modest risk that flooded packets would fail to reach some nodes. Compensation packets could then be used to overcome this low level of residual losses.

3. MISTRAL

Traditional flooding suffers from the problem of redundant message reception, once per neighbor. Even in a reasonably connected network, the same message is received multiple times by every node, which is inefficient, wastes valuable resources, and can create contention in the transmission medium.

Selective rebroadcasting of flooded messages is a way to limit the number of redundant transmissions. Instead of simply rebroadcasting the message a node evaluates a local function $\mathcal{F}$ and then uses the outcome of this computation to decide whether to forward the message. In its simplest form, this function returns its result based on some static probability (corresponding to PPF). More complex functions take into account additional topological (e.g., the number of neighbors) or statistical information (e.g., the number of overheard rebroadcasts). The downside of selective flooding is that a flooding may no longer reach all intended nodes. In particular, if a node has only few neighbors, none of these neighbors may rebroadcast the message. Selective flooding thus balances message overhead against reliability.

Mistral finds some middle ground by introducing a new mechanism that allows us to fine-tune the balance between message overhead and reliability. The key idea is to extend selective flooding approaches by compensating for messages that are not rebroadcast. This compensation is based on a technique borrowed from forward error correction (FEC). Every incoming data packet (dp) is either rebroadcast or added to a compensation packet (cp). The compensation packet is broadcast at regular intervals and allows the receivers to recover one missing data packet.

3.1 Forward Error Correction

In its simplest form, Forward Error Correction (FEC) [11,

19, 22] creates l repair packets for every m data packets such that any m out of the resulting $(m+l)$ packets is enough to recover the original m data packets [11]. Traditional applications of FEC generate l repair packets for every m data packets and inject them into a data stream, which insulates the receiver from at most l packet losses. One of the fundamental advantages of FEC is that it imposes a constant overhead on the system and has easily understandable behavior under arbitrary network conditions. However, this simple form of FEC was developed for streaming settings, where a single sender is transmitting data at a high, steady rate such as in bulk file transfers [6] or in a video or audio feeds [7]. Part of our challenge is to develop a FEC solution matched to the characteristics of a MANET.

3.2 Algorithm

We noted earlier that Mistral can be built on top of any local-knowledge-based flooding approach. In the current implementation of the system, we use purely probabilistic flooding, mostly because this approach is extremely simple and is intuitively easy to visualize. Recall that in PPF, a node rebroadcasts a flooded message with static probability p. Although PPF might not be an ideal choice of algorithm in a practical deployment, the algorithm has no "hidden" effects that might make it hard to interpret our experimental findings.

Upon reception of a data packet, every node evaluates the function $\mathcal{F} : dp \longmapsto \{true|false\}$. In its most basic form, $\mathcal{F}$ takes a data packet as input and returns a boolean. If it returns true, dp is rebroadcast; otherwise, dp is added to the current compensation packet. When the number of data packets contained in a compensation packet passes a certain threshold c, the compensation packet is broadcast. We call c the *compensation rate*. Thus, a compensation packet is broadcast for every c data packets that are not rebroadcast.

Algorithm 1 presents the algorithm in more detail: Procedure `process` delivers the data packet to the application and decides whether to rebroadcast the packet or add it to the compensation packet; `composeCompensationPac` builds the compensation packet; and `runRecovery` attempts to recover data packets from stored compensation packets when a new data packet is delivered to the application. Finally, procedure `expand` is used for level-2 recovery, which is presented in Section 3.2.2. The secondary recovery mechanism discussed in the introduction is not included in Algorithm 1.

3.2.1 Composition of a Compensation Packet

In this section, we assume that data packets are of fixed size, e.g., 512 bytes, and contain the payload, a sender ID and some locally unique sequence number; we call these the *packet id*. The payload is assumed to remain unchanged during the course of the flooding (in some protocols, payloads do change as packets are routed; we discuss the handling of this kind of mutable payloads later in the paper).

To encode the payload of the data packets into the compensation packet, we use the XOR (operator $\otimes$), which is the simplest and best known FEC mechanism. A new data packet is added to the compensation packet by computing the XOR of its payload with the current payload in the compensation packet (initially, zero). Obviously, much more sophisticated error correction mechanisms are also possible; the advantage of XOR is its simplicity and low computational overhead.

Algorithm 1 Mistral's algorithm, code of node n_i.

```
1: Initialisation:
2:   DpBuffer ← ∅                                   {Received dps}
3:   cp ← ⊥                                         {Compensation packet}
4:   CpBuffer ← ∅                                   {Received cps}

5: upon flood(dp) do
6:   broadcast(dp)

7: upon reception of data packet dp for the first time do
8:   process(dp)
9:   runRecovery(dp)

10: upon reception of compensation packet cp from sender p_j
    do
11:   if cp.ids contains unknown dp ID then
12:     if recovery possible then
13:       dp_recov ← recover from cp
14:       process(dp_recov)
15:       runRecovery(dp_recov)
16:     else
17:       CpBuffer ← CpBuffer ∪ {cp}
18:       if level-2 recovery then
19:         expand(cp)
20:         for all recovered dp do
21:           process(dp)
22:           runRecovery(dp)

23: procedure process(dp)                  {handles a data packet}
24:   DpBuffer ← DpBuffer ∪ {dp}
25:   if F(dp) then
26:     broadcast(dp)
27:   else
28:     composeCompensationPac(dp)
29:   deliver dp to the application

30: procedure composeCompensationPac(dp)   {constructs a
    cp}
31:   cp.payload ← cp.payload ⊗ dp.payload
32:   cp.ids ← cp.ids ∪ {dp.id}; cp.ttls ← cp.ttls ∪ {dp.ttl}
33:   if |cp.ids| ≥ c then{|X| returns the nbr of elements in
      X}
34:     broadcast(cp)
35:     cp ← ⊥

36: procedure runRecovery(dp)              {recovers dps from
    CpBuffer}
37:   for all cp1 ∈ CpBuffer do
38:     if dp.id ∈ cp1.ids then
39:       remove dp from cp                {including TTL and ID}
40:       if recovery from cp1 possible then
41:         dp_recov ← recover from cp1
42:   for all recovered data packets dp'_recov do
43:     process(dp'_recov)
44:     runRecovery(dp'_recov)

45: procedure expand(cp)                   {level-2 recovery}
46:   for all cp1 ∈ CpBuffer do
47:     for all cp2 ∈ CpBuffer ∧ cp2 ≠ cp1 do
48:       if cp1 or cp2 is reducible then
49:         cp ← reduction from cp1 and cp2
50:         CpBuffer ← CpBuffer ∪ {cp}
```

If the receiver of a compensation packet already has all but one of the contained data packets, the compensation packet will allow the reconstruction of that missing data packet. However, the recipient of a compensation packet has no a-priori way to know what data packets were used to build the compensation packet. Accordingly, compensation packets must include a list of all its contained data packet IDs. Assuming IP-style node addresses, the sender ID is represented by four bytes. The local sequence number consists of one byte, which allows Mistral to send 255 flooded messages by a node before looping back to 0. From this, we can see that the size of a compensation packets will be the payload size plus five times the number of included data packets c, i.e, $|cp| = |payload_{dp}| + 5 * c$. Notice that the packet size is independent of the number of nodes in the system as a whole. This information is sufficient for floodings that span the entire node field.

A complication arises in applications where the the scope of flooding is limited by a time-to-live (TTL) parameter. Here, the compensation packets need to represent the TTL for each contained data packet; otherwise, if a node recovers data packet dp from a compensation packet, it has no way to know what TTL to use when rebroadcasting dp. If it chooses a TTL that is smaller than the true TTL, then the flooding may die out too early. If the TTL is too high, then valuable bandwidth is wasted. Even worse, if the flooding is a part of a routing mechanisms and the routing mechanism depends on the TTL, then loops occur in the routing paths.

Clearly we cannot treat the TTL of a data packet as a part of that packet's payload, since TTLs are decremented at every hop of the data packet. The problem here is that incoming TTLs for received packets might differ at the node undertaking the reconstruction relative to the node that built the compensation packet. Thus, TTLs need to be added to the compensation packet outside of the payload.

The simplest approach is to add a list of TTLs to the compensation packet. Since the TTL is generally represented by one byte another c bytes are added to the size of a compensation packet. In effect, the TTL extends the packet-id by one byte.

Unfortunatly, this simple approach adds additional overhead, which we would like to avoid. A first point to notice is that TTLs are often defined based on some estimate and are thus, by design, already an approximation. Hence, if we manage to limit the error to some low number, we can manage with an approximate reenstruction of the TTL value. For instance, we could store the sum of all TTLs. The TTL of a recovered data packet can then be restored by subtracting the TTL's of all known packets (all data packets except one). To limit the size to one byte, we apply the modulo operator to this sum. Using this approach, the error will in most cases be within ± 1, or in total $\pm c$, which is acceptable for most applications. Thus, the total size of a compensation packet is $5c + 1 + |payload_{dp}|$ bytes.

Although we have not explored the idea yet, it may be possible to further reduce the overhead associated with compensation packets by compressing packet-id information. For example, in a MANET where most communication originates with a very small set of senders, we could assign those senders some sort of very small id. Moreover, it may sometimes be possible to compress the compensation packet payload itself. On the other hand, such ideas increase the computational overhead at the receiver and hence would require careful evaluation.

3.2.2 *Recovering from Compensation Packets*

To recover data packets from compensation packets we use a two-level recovery mechanism. The first level recovers data packets based on the data packets that have already been received. If $c - 1$ data packets contained in a compensation packet are known, the missing one can be reconstructed. Compensation packets that contain two or more missing data packets are stored (in the $CpBuffer$) and reconsidered when new data packets arrive or are recovered from other compensation packets. Actually, we do not store complete compensation packets, but only compensation packets that contain the IDs, TTL(s), and payload of the missing packets. More specifically, we xor the known data packet payloads with the payload of the compensation packet. After some time compensation packets are garbage collected, as it has become highly unlikely that the missing data packet(s) will be received in the future.

The level-2 recovery mechanism is more elaborate. Instead of only considering incoming and recovered data packets this algorithm also matches compensation packets against each other. The matching operation works as a reduction. Each new compensation packet is compared with all stored compensation packets. If either one of the packets is completely contained in the other, then a new compensation packet is added, which contains the set of data packet IDs of the larger packet minus the ones in the smaller packet. The new payload is constructed by applying XOR to both compensation packets. Provided that it does not allow the immediate recovery of a data packet, this reduced compensation packet is then added to the set of stored compensation packets (in $CpBuffer$).

Clearly, level-2 recovery adds a considerable overhead, both in storage and computation. Its application thus makes sense only if the gain in recovered data packets is significant with respect to level-1 recovery. We explore level-1 and level-2 recovery using simulations in Section 5.2.3.

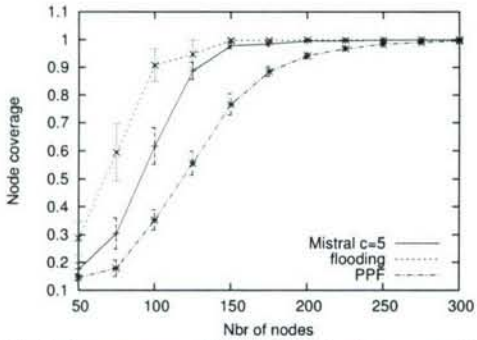
3.2.3 *Mutable Payloads*

Many routing protocols modify the flooded packets during the flooding. We have already shown how to handle TTL values. But some protocols modify other parts of data packets, for example by touching internal parameters, building a route trace, etc. To allow Mistral to handle these cases, we extend the above mechanism into compensation packets that include a mutable part and an immutable part of the payload. Clearly, the larger the immutable part is relative to the mutable part, the better the performance of Mistral. This is particularly the case as the immutable parts can be reduced into an immutable part of the same size, while mutable parts need to be appended to each other, thereby resulting in a size of $\sum_{i=0}^{c} mutablePartOf(dp_i)$. In genernal, the size of a compensation packet will now be $5c + |immutablePayload_{dp}| + \sum_{i=0}^{c} mutablePartOf(dp_i)$.

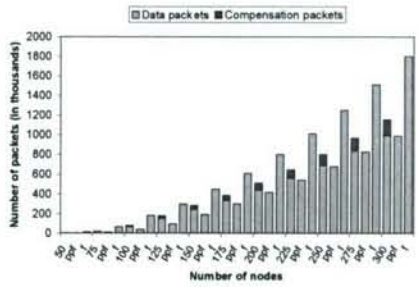
In the evaluation that follows, we assume that packets contain no mutable data other than the TTL.

4. ANALYSIS

In this section, we provide a simple analysis of Mistral. We denote by d_{max} the maximal diameter of the node field and consider floodings that span the entire node field. The maximal transmission latency $t_{maxTrans}$ is the maximal trans-



(a) Node coverage with varying density, $p = 0.55$.



(b) Message overhead with varying density, $p = 0.55$.

Figure 1: Node coverage and message overhead with varying node density.

mission range (88m) divided by the transmission speed. The time needed to do all the computations on a node is Δt, and we assume that there are no delays in the outgoing sending buffers, i.e., that there is no contention in the transmission medium.

Let f_i denote the number of floodings originating at node i, then the estimated overall generated number of compensation packets in a network with n nodes is $G = n\frac{(1-p)\Sigma_i f_i}{c}$, assuming that every node receives all flooded data packet at least once. Thus, the overhead in bytes is $G * (5c + 1 + |payload_{dp}|)$.

Assume that δ_{flood} denotes the average reception frequency of data packet that are received for the first time. Then, the estimated time needed to fill up a c-based compensation packet is $t_{recoveryPac} = \frac{c}{(1-p)*\delta_{flood}}$.

We now consider the delivery latency of a data packet. The worst case occurs when the flooding source and the destination are d_{max} hops apart and the data packet is always forwarded as part of a compensation packet. In this case, the maximum delivery latency is $d_{max} * (t_{recoveryPac} + \Delta t + t_{maxTrans})$, while the estimated maximum delivery latency is $(1-p) * d_{max} * (t_{recoveryPac} + \Delta t + t_{maxTrans}) + p * d_{max} * (\Delta t + t_{maxTrans})$.

We now compute the number of packets sent by a single flooding in a network of N connected nodes. Purely probabilistic flooding has a message overhead of $E(MsgOverhead) = p * N$, if we assume that every node receives the flooded message at least once. Mistral adds an estimate of $\frac{1}{c}$ for every dropped message. Thus, the total overhead per flooding is $(1-p) * N * \frac{1}{c} + p * N$. If the assumption that all nodes receive the flooded message is relaxed then the relative overhead added by Mistral increases. Each node that receives the flooded message only because of Mistral again contributes an additional broadcast or partial compensation to the overhead. Naturally, the additional overhead pays off through the increased node coverage.

5. SIMULATIONS

For our simulation we used JiST/SWANS v1.0.4 [1, 4], a simulation environment for ad hoc networks. Java applications written for a real deployment can be ported to the simulation environment and then placed under a variety of simulated scenarios and loads. JiST/SWANS intercepts the calls to the communication layer and dynamically transforms them into calls to the simulator's communication package.

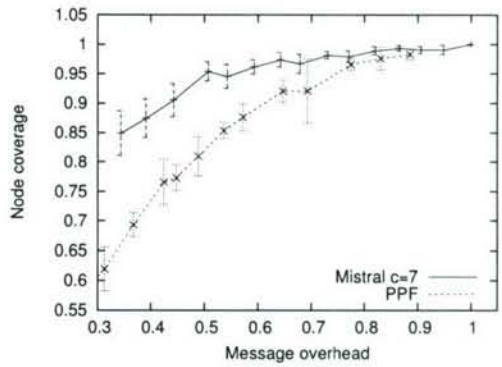
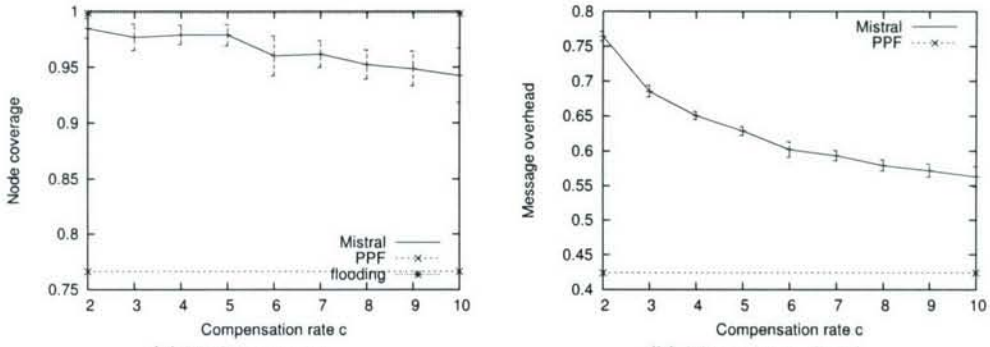


Figure 2: Node coverage with respect to message overhead.

5.1 Setup

We consider a set of nodes. Communication between two nodes m and n occurs in an ad hoc manner and may be asymmetric, i.e., n may be able to communicate with m, but the inverse may not be possible. Communication is by broadcast as defined in the 802.11b standard [12] and can be subject to interference, in which case the message cannot be received. Interference can occur without the sender being able to detect the problem (this is called the *hidden terminal problem* [2]).

We simulate a wireless ad-hoc network with 150 nodes uniformly distributed in a field of size 600x600m. Nodes are stationary, except for one case in which we measure the impact of mobility (Section 5.2.4). The maximal transmission range of a node is set to 88m. Every node starts flooding 20 messages at a regular interval, once all nodes are started up. All flooding occurs across the entire node field. Hence, ideally all nodes should receive all flooded messages.



(a) Node coverage. (b) Message overhead.

Figure 3: Varying compensation rate c, $p = 0.55$.

Our work models disconnections due to mobility, transmission range limits, and the hidden terminal problem just mentioned (using JiST/SWANS' *RadioNoiseIndep* package, which uses a radio model identical to ns2). Unless otherwise mentioned, we use the default values defined in JiST/SWANS.

The nodes start up at random times and positions. When they are all up and running, we start sending the flooding messages and we wait until all messages have been received (terminating simulation).

5.2 Results

In this section, we present the results of our simulation. Every node periodically, every 50s, floods a message throughout the entire field. We have chosen a low flooding rate because in our simulations we want to minimize the effect of packet loss due to buffer overflows and interference. The nodes are added to the sensor field at time 0s but start flooding at times uniformly distributed between 0 and 60s. All results give the average over at least 30 runs in different uniform node distributions. In general, the variance in the simulation results for ad hoc networks is high. This is due to the many sources of randomness: distribution of the sensor nodes, the paths of nodes, the time the nodes flood a message, etc. Thus, where significant we indicate the 95%-confidence intervals (CI).

To evaluate the quality of Mistral, we are mainly interested in two properties: node coverage and message overhead. *Node coverage* measures the number of nodes that have received the messages, while message overhead indicates the total number of sent messages. Both measurements are normalized against a connected network with the same number of nodes. In a connected network, any node can communicate with any other node. Thus, node coverage is given as a percentage of all nodes in the network, while message overhead is given as the percentage of the message overhead in the case in which all nodes receive all messages (normal flooding). Note that the message overhead in the connected network equals the product of the number of flooded messages with the number of nodes. Moreover, it is generally lower in a network with partitions. Since Mistral complements local-knowledge-based approaches and is based on purely probabilistic flooding, we compare Mistral to the latter. Purely probabilistic flooding is entirely defined by the rebroadcasting probability p. For completeness, we also show the results for simple flooding, which corresponds to PPF with $p = 1.0$.

In the following, we evaluate the following properties of Mistral: its behavior in the face of varying density, varying protocol parameters, node mobility, packet loss, and with the secondary recovery mechanism. Unless explicitly stated otherwise, we use the above default values in our measurements.

5.2.1 *Impact of Density*

We start by measuring the impact of node density on the node coverage and the message overhead. Fig. 1(a) shows the node coverage with varying number of nodes. It shows three measurements: simple flooding, purely probabilistic flooding (PPF), and Mistral with compensation rate $c = 5$. The rebroadcast probability is set to $p = 0.55$ in the cases of purely probabilistic flooding and Mistral. As expected, Mistral has a much higher node coverage than purely probabilistic flooding, especially for lower node densities. If the node density passes a certain threshold (around 225 nodes for Mistral), it is sufficiently high such that all nodes receive all messages. In contrast, with low density only a low percentage of the nodes receive all messages. However, below a certain threshold (around 150 nodes) even simple flooding cannot reach all nodes.

In Fig. 1(b) we show the corresponding message overhead. For every number of nodes indicated on the x-axis, we draw the sent number of packets for Mistral, purely probabilistic flooding (ppf), and simple flooding (f). Mistral's packets are further separated into data packets and compensation packets. Since Mistral adds additional compensation packets, its total message overhead is higher than the one of purely probabilistic flooding. Notice also that for low densities the number of flooding packets is higher. Due to higher node coverage in Mistral, more nodes receive the message and thus more nodes also rebroadcast the message, which accounts for the higher number of flooding packets compared to PPF.

Thus, to measure Mistral's net gain in node coverage, as compared to purely probabilistic flooding, we need to consider both node coverage and message overhead graphs. Indeed, since Mistral's compensation mechanism adds an additional overhead, we cannot directly compare the two approaches with the same rebroadcast probability p. Rather,

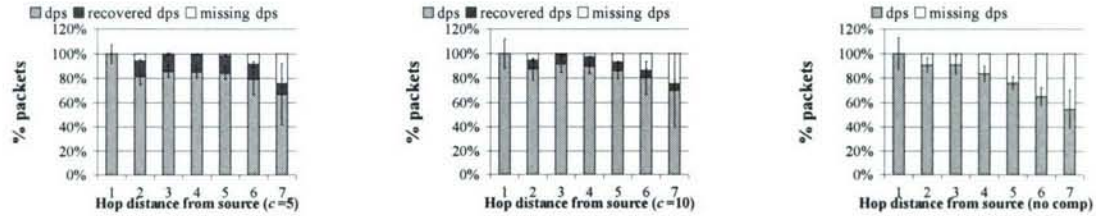


Figure 4: Recovery based on hop counts, single source and $p = 0.55$.

we need to compare Mistral with the purely probabilistic flooding using a rebroadcasting probability with a similar message overhead. Fig. 2 plots the node coverage with respect to the message overhead, for $c = 7$. The message overhead corresponds to simulation runs with p varying from 0.3 (0.4 for PPF) to 1, in steps of 0.05. The gain with Mistral is especially prominent for low rebroadcast probabilities p. Of course, low rebroadcast probabilities lead to many dropped rebroadcasts and thus the node coverage becomes low. Using Mistral allows some of the nodes to recover messages they may have missed. For an overhead of 0.35, Mistral improves the node coverage by 20%, for an overhead of 0.55 by 10%, and for overhead around 0.75 it is closer to 3%.

5.2.2 *Compensation Rate*

We now turn to one of the parameters that determine the behavior of Mistral: compensation rate c. In Fig. 3(a), we show the node coverage with compensation rate c varying from 2 to 10, for 150 nodes and rebroadcasting probability $p = 0.55$. Generally, the node coverage decreases with increasing compensation rate. For comparison, the graph also indicates the node coverage for flooding and purely probabilitstic flooding (PPF) with the same parameters. Both flooding and probabilistic flooding are independent from the compensation rate and thus are represented by a horizontal line. Fig. 3(b) gives the corresponding message overhead. Here, the message overhead decreases with increasing compensation rate. Thus, given a particular node coverage the higher the compensation rate the better. However, a higher compensation rate also increases the message delivery latency. Indeed, data packets that are part of a compensation packet spend more time waiting until the compensation packet is filled with sufficient data packets and may thus be delayed.

5.2.3 *Recovery Performance and Overhead*

Next, we measure the number of recovered data packets with respect to the hop count (see Fig.4). In this simulation, a single node at position $[300, 300]$ periodically floods 1000 messages. We give the results for $c = 5$, $c = 10$, and the case with no compensation (no comp). Since the overall number of received data packets is different depending on the hop-distance of a node to the flooding source, we give the percentage of recovered data packets to all flooding packets that should have been received by the nodes at this hop distance from the source. The percentage of recovered data packets is approximately the same for most hop distances. An exception is at hop count 1, where all nodes generally receive the flooded message, because the source floods the data packet with $p = 1.0$ and at a time of low traffic. As fewer compensation packets are sent in the case of $c = 10$, the percentage of recovered data packets is lower compared to the case of $c = 5$. Towards very high numbers of hop counts, no compensation packets are received. However, these nodes are particular cases resulting from a unusual node distribution, which does not occur frequently.

Notice that the percentage of dps increases between $c = 5$ and $c = 10$. The reason is that with smaller c, more compensation packets are sent and the likelihood that a dps is received via a compensation packet increases. Since we count the data packets when they are received for the first time, more packets are received via a compensation packet. Thus, the percentage of dps is smaller for a smaller c.

We use the same setup to measure the packet delivery latency. In contrast to the other simulations, we use a single data point (one random uniform distribution) in this case. The single source floods a data packet every second. The graphs in Fig. 5(a) and (b) show the latency distribution of data packets for $c = 2$ and $c = 8$ with respect to the hop distance of the node. The delivery latency of a data packet is high if it is received by a node only as part of a compensation packet. The higher the compensation rate, the higher this delay is.

Another important characteristic of Mistral is the ratio of compensation packets that cannot be recovered. We say that a *compensation packet is recovered* if all contained data packets have been received or have been recovered. In general, we expect the number of unrecovered compensation packets to increase with increasing compensation rate. The graph in Fig. 5(c) confirms this. It uses our default setup with many flooding sources and shows the total number of received compensation packets and the number of compensation packets that have not been recovered (logarithmic scale on y-axis). Clearly, the lower the compensation rate, the higher the number of sent and thus received compensation packets. This number also includes all compensation packets whose contained data packets have already been received earlier by the receiving node (useless cps). Immediately recovered compensation packets denote the compensation packets that only contain a single unknown data packet. Any compensation packet that contains more unknown data packets is added to *CpBuffer*.

Fig. 5(d) shows the number of sent compensation packets based on the number of nodes in the field. As expected, this

Scalable Technology for a New Generation of Collaborative Applications

MURI Grant Final Report
AFOSR F49620-02-1-0233
May 2002 – April 2007

1. Principal Investigators

Ken Birman (Cornell University)
Al Demers (Cornell University)
Johannes Gehrke (Cornell University)
Keith Marzullo (University of California at San Diego)
Geoffrey M. Voelker (University of California at San Diego)

2. Research Objectives

Our MURI effort emerged from dialog between the AFRL team developing software for the Joint Battlespace Infosphere (JBI) and university researchers at Cornell and elsewhere. It became clear that to be successful, the JBI needed to break completely new ground in offering publish-subscribe capabilities on a scale never previously attempted, and do so with guarantees of security, reliability and predictable performance of a sort impossible for existing commercial products. The AFRL JBI team reacted to this unique challenge by evaluating a number of candidate "core" technologies for distributed systems that map in clear ways to the technical needs of the JBI:

- Group communication (multicast) systems and commercial publish-subscribe systems have a direct correspondence to the communications needs of the JBI.
- The JBI repository will be a database that can be updated and queried in real-time, and there are thus parallels between repository functionality and commercial products in the database area.
- Because many JBI information sources provide data streams or periodic data bursts (indeed, few are likely to be static), there is a strong connection between JBI reporting functionality and the technology of distributed data mining and triggered actions.
- The JBI will use off-the-shelf Web Services technologies wherever possible, thus the degree of match between the JBI and such systems must also be better understood.

It quickly became clear that no existing commercial product is adequate for the full spectrum of JBI requirements. While a number of prototypes have been constructed for various components of the JBI using readily available technologies, these integrate poorly and lack the distributed computing functionality and scalability properties required of the real system. On the basis of commercial experience, any JBI system built in this manner will be fragile and easily disrupted under stress, and omit

important functionality (such as integration of the publish-subscribe aspects of the JBI and the repository) that are lacking in existing commercial offerings. Accordingly, AFRL encouraged the research community to look both at the technical needs of the JBI and its sibling systems in the Navy, Army and Marines, and also to work towards the elaboration of a scientific basis for reasoning about systems such as the JBI - a science some are dubbing "Infospherics" because of its applicability to the Infosphere.

Particularly important are demonstrations of scalable solutions which can be counted upon to remain stable under stress and to offer predictable performance and reliability even when the system is disrupted by an adversary while performing mission-critical tasks. These needs extend from the publish-subscribe functionality per-se to other aspects, such as querying data residing in networks of sensors. As the JBI is adapted for use by other services, some of these aspects may require urgent attention. For example, the Navy is considering the JBI as a platform for future USW sensor applications, but these are primarily data mining problems, not publish-subscribe applications. Understanding how to make such solutions coexist in a single platform is vital for the JBI to emerge into the desired role for the Air Force and its sibling military services.

At Cornell and the University of San Diego, our team came together to pursue common interests at the intersection of large-scale distributed computing systems, Web Services and similar emerging architectures, data mining, and distributed database systems. Our group spans the technology areas needed by the JBI and is united by a shared interest in similar kinds of technologies (particularly probabilistic approaches with good scaling properties), similar application models (relating to publish-subscribe, data mining, and other forms of communication patterns best described as forms of query evaluation), and extensive expertise in experimental evaluation of the technologies we develop.

Birman and van Renesse jointly head the Spinglass group, which is a broad effort developing a new family of reliable, scalable protocols for communication, monitoring, management and control in large distributed systems. Spinglass exploits "multipeer" communication to achieve scalable, probabilistically reliable solutions to component problems that arise in a variety of setting. These components can then be used as tools for enabling direct, live collaboration between participants who may be spread across a global network and using platforms with differing communications capabilities.

Birman and his group focused on scalable group communication systems that provide critical properties such as data replication, distributed coordination, and automated reaction to faults. They developed systems that provide these properties and scale publish-subscribe and group multicast services in many dimensions. They also developed tools for transforming standard Web services into ones that scales to clusters in data centers, automatically replicates data to improve peformance and reliability, and efficiently updates replicated data. Van Renesse and his group targeted the problem of hardening group communication systems to failures and attacks. They developed reliable techniques for detecting intruders in group communication systems and mitigating the damage that they can cause, such as providing incorrect information or dropping packets.

Gehrke and Demers investigated scalability and information management issues in the JBI and other military information dissemination systems. They worked on the challenges of building very large scale

databases which are spread among many sites and yet maintain strong forms of consistency and can be queried in a manner similar to the ways that centralized databases can be queried. They targeted emerging architectures where actual databases reside at the leaves, are updated using transactions, and can support triggered upcalls when events change in ways of interest. Within this overall effort, Gehke's group has been focused on extending existing database systems and building completely new ones optimized for use in distributed sensor networks and other systems that generate high volumes of data rapidly. Demers focused on understanding the behavior of systems built using probabilistic protocols and the risks associated with using approximate algorithms, developing a number of such algorithms for solving simple problems like solving distributed range queries over complex sensor networks, computing aggregate values in large systems, and optimal resource location in distributed settings.

Marzullo and Voelker in the UCSD group focused on providing high service availability and efficient reliability for large-scale distributed systems that form the foundation of JBI efforts. Voelker and his group addressed the problem of providing highly-available services in distributed systems composed of relatively unavailable components. A fundamental challenge for designing and building next-generation distributed systems is providing high availability using these highly unavailable components. Their work developed middleware that provides an abstraction of a highly-available platform to upper-layer software services while running on intermittently available components. They demonstrated this approach in a prototype wide-area file system. In terms of reliability, Marzullo and his group pursued the development of and insights into the theoretical understanding of distributed systems with dependent failures. Their work has developed new solutions to several well-known problems in distributed computing that are optimal when failures are not independent and do not have identical distributions. Making their theoretical results practical, they applied dependent failure system models to a system called Phoenix that cooperatively protects data from loss arising from Internet catastrophes from propagating malware such as viruses and worms.

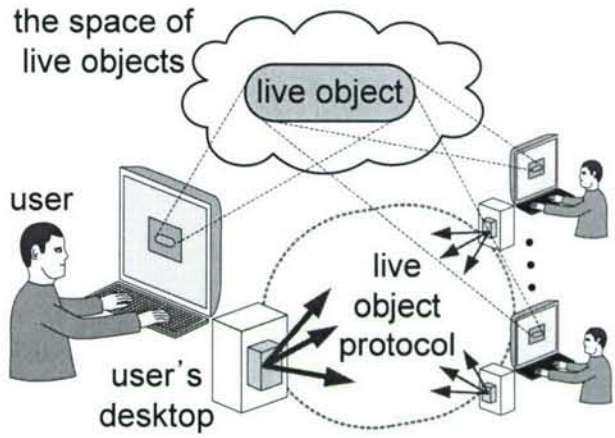
The JBI has proved to be an outstanding focus for our collaboration, and we embarked on a research agenda to use the MURI funding, together with other funding available to us directly from the JBI community and from other sources, such as DARPA, to assist AFRL in taking a major step forward on these challenging and extremely important questions. Over the period of the MURI grant, we established substantial groundwork for a scientific treatment of the military's pressing challenges, backed by rigorous experimental work that helps clarify the uncertainty concerning just how a JBI system might actually be implemented. We believe that the research results from this MURI effort will thus pave the way for future commercial efforts to provide concrete deliverables to the JBI and similar military projects in other services.

3. Status of Efforts, Accomplishments, and New Findings

To better encapsulate the major research accomplishments performed by our group, we next provide individual summaries of the goals, approaches, accomplishments, and impacts on military needs of our efforts funded through this MURI effort.

QUICKSILVER SCALABLE MULTICAST

Goal: *GIG/NCES platforms provide excellent support for point-to-point communication in a web services paradigm, but support for scalable group communication, data replication, distributed coordination and automated reaction to faults are sorely lacking. Our goal in the Quicksilver project is to overcome the inherent scalability problems that limited development of solutions having these properties. The basic premise is that if we can show how to scale a publish-subscribe or group multicast system in many dimensions, in a web services framework, we can then take the next step and build the missing tools.*



Approach: *We developed Quicksilver Scalable Multicast, and found a new way to embed the technology into the Windows and Linux platforms based on what we call a "live objects" interface. The key ideas were as follows:*

- *A live object extends the normal Windows support for component integration to permit a new kind of component in which members belong to a group that replicates data using high-speed multicast. Various back-end communication "drivers" can provide the multicast; QSM is just the first of these*
- *We constructed an implementation of such a multicast protocol, QSM, and have demonstrated that it can scale far beyond the limits of any prior multicast technology, particularly when large numbers of users employ the system and this results in a pattern of extensively overlapped multicast groups. QSM maintains extremely high performance even under disruptions and other forms of injected stress.*
- *We've begun to extend QSM with a new high-level language so that reliability can be layered over the core system in a simple, easily used manner that can support everything from best-effort reliability to strong models such as virtual synchrony or transactional one-copy serializability.*

Accomplishments: *Our work on QSM is attracting attention from major vendors such as Cisco and Microsoft, even as we develop a small user community of early adopters. With the expected release of our live objects layer in the fall of 2007, we should see a burst of users drawn to our system by the ability to create live documents and applications by dragging and dropping live objects onto web pages, into databases, or in Word documents. The basic idea is that by creating such a document and then sharing it, non-programmers can create sophisticated distributed applications much as they build PowerPoint presentations.*

We are also finding that the Web Services standards community is interested in our approach. A journal article proposing a way to extend WS-NOTIFICATION and WS-EVENTING to offer greater flexibility will appear in the fall.

Our work on Quicksilver will continue beyond the termination of the MURI effort under funding from AFRL, AFOSR and other sources.

Impact on the warfighter: Quicksilver enables a new kind of agile response to rapidly evolving conditions. Today, it would be impossible to imagine building, say, a customized application for a search-and-rescue mission on the fly, in the field. With live objects and Quicksilver it may be possible for tomorrow's commander to create custom information solutions as needed, in real-time, for instant deployment to the troops for whom accurate timely information can determine mission success.

References: (For downloads and complete list, visit http://www.cs.cornell.edu/projects/quicksilver/)

Scalable Publish-Subscribe in a Managed Framework. Krzysztof Ostrowski, Ken Birman. Cornell Technical Report (TR2007-2086). October, 2006.

The QuickSilver Properties Framework. Krzysztof Ostrowski, Ken Birman, Danny Dolev. OSDI'06 poster session, Seattle, WA, November 2006.

Properties Framework and Typed Endpoints for Scalable Group Communication. Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report TR2006-2062 (July, 2006).

Scalable Group Communication System for Scalable Trust. Krzysztof Ostrowski, Ken Birman. In Proceedings of The First ACM Workshop on Scalable Trusted Computing (ACM STC 2006). Fairfax, VA. November 3, 2006.

Extensible Web Services Architecture for Notification in Large-Scale Systems. Krzysztof Ostrowski and Ken Birman. In Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006). Chicago, IL, September 2006

Declarative Reliable Multi-Party Protocols Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Implementing High-Performance Multicast in a Managed Environment Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems. Ken Birman, Anne-Marie Kermarrec, Krzysztof Ostrowski, Marin Bertier, Danny Dolev, Robbert Van Renesse. Distributed Event Processing Systems and Architecture Workshop (DEPSA). June 2007.

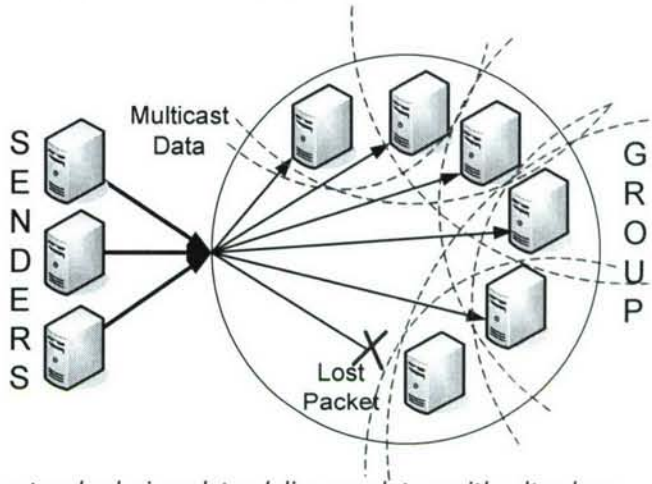
Live Distributed Objects: Enabling the Active Web Krzysztof Ostrowski, Ken Birman, Danny Dolev. To Appear in IEEE Internet Computing.

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. To Appear in the International Journal of Web Services Research. Volume 4, Number 4. October-December 2007.

RICOCHET SCALABLE TIME-CRITICAL MULTICAST PROTOCOL

Goal: *Data center systems are challenged by the difficulty of rapidly disseminating time-critical information, for example within a service running on multiple nodes in a cluster and where the data will trigger some real-time response. Such problems arise in many settings, including radar tracking systems, weapons targeting, real-time control of autonomous vehicles, etc. Moreover, there are many settings in which "standard" web services need to offer rapid end-user response time, even as updates flow into the core system.*



Approach: *Ricochet/Plato are a new family of multicast protocols designed to deliver updates with ultra-low latencies in clusters or data centers hosting scaled servers, again under the assumption that the services are implemented using web services standards. The key idea here is to aggregate traffic so that error correction codes can be computed more quickly than if each data stream was treated separately. Ricochet and Plato explore this in a multicast setting; we are currently taking the next step by developing, Maelstrom, which applies similar ideas with an emphasis on connections between data centers over long-distance WAN links. Cisco and Microsoft have shown keen interest in using these solutions in their respective platforms and products, and Raytheon is helping us explore transition into military platforms using the DDS standard.*

Accomplishments:

- *Designed and implemented the Ricochet protocol, undertook a comprehensive evaluation, and completed a series of papers on this work, including mobile wireless applications (Mistral).*
- *Through dialog with companies in the web services community, identified promising technology transition opportunities. Raytheon is taking the lead on pursuing these with us, focusing on the military-standard DDS communication architecture.*
- *Made our solutions available to other vendors, including the Apache web services platform.*
- *Developed and implemented a predictive real-time ordering protocol, Plato.*

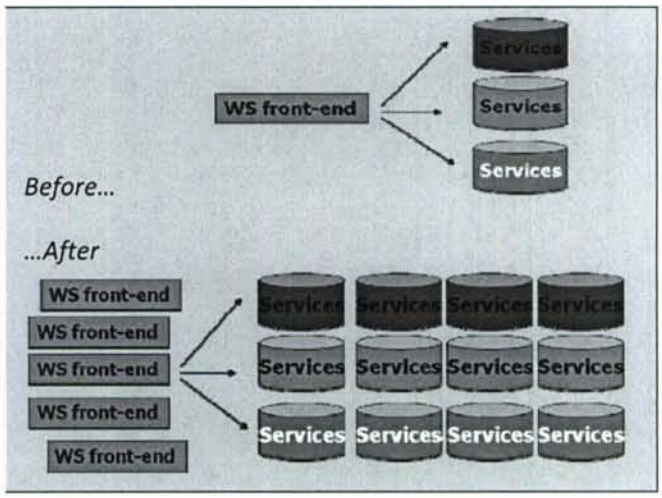
Impact on the warfighter: During military operations, timely information can have life or death implications. Ricochet makes possible dramatically improved responsiveness and maintains this guarantee as it scales up.

References: *Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*

Ricochet: Lateral Error Correction for Time-Critical Multicast. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. To Appear in Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). Cambridge, MA. April 2007.
Mistral: Efficient Flooding in Mobile Ad-hoc Networks. S. Pleisch, M. Balakrishnan, K. Birman, and R. van Renesse. In Proceedings of the Seventh ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM MobiHoc 2006). Florence, Italy May 2006.
PLATO:Predictive Latency-Aware Total Ordering. Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. In Proceedings of the SRDS 2006: 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK. October 2006.

TEMPEST: A TOOL FOR CREATING SCALABLE WEB SERVICES

Goal: *Today it is much too difficult for programmers with a typical MEng-level of training to implement scalable, self-managing web services that can run on datacenters in GIG/NCES settings and that will automatically adapt as conditions change. By solving this problem we can reduce the delays and costs associated with implementing new services for the GIG while also ensuring that those services will be seamlessly adaptive even under stress. Moreover, we can bring best-of-breed solutions to the table in a reusable form, reducing the tendency of vendors to produce stovepipe technologies that only the developer can extend or support.*



Approach: Tempest is a tool for turning a fairly vanilla web service into one that scales on a cluster or data center, has automatically replicated data, and employs Ricochet to send updates. Gossip communication is employed as an adjunct to this to repair any inconsistency that might arise as a result of a failure. Tempest is just reaching a stage at which early demos are feasible; the system is able to take a front end (for example an application that builds web pages) and a set of back end web services and will automatically replicate each of these, to varying degrees, in a manner that achieves predictable time-critical response even under stress, even when faults occur, and even when the services have very different behavioral profiles (such as mean response time, etc).

Accomplishments: Tempest was completed in early 2006 and works well; it uses a novel gossip-based approach to propagate updates, detect and repair inconsistencies, and for self-management of the clustered web application. In current work, we are exploring an extension in which Tempest could be used to create integrated enterprise web applications, still in a highly automated manner, with our gossip protocols used to extract data from source applications, replicate it across a WAN, and then integrate it with an application needing to import that data remotely.

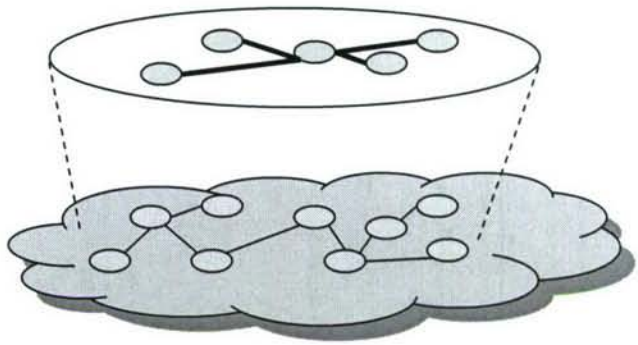
References: *Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

A Scalable Services Architecture. Tudor Marian, Ken Birman, and Robbert van Renesse. To appear in Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2006). Leeds, UK. October 2006.

Fireflies: Scalable Byzantine Overlay Networking

Goal: *"Fireflies" is a scalable protocol for supporting intrusion-tolerant network overlays. While such a protocol cannot distinguish Byzantine nodes from correct nodes in general, Fireflies provides correct nodes with a reasonably current view of which nodes are live, as well as a pseudo-random mesh for communication. The amount of data sent by correct nodes grows linearly with the aggregate rate of failures and recoveries, even if provoked by Byzantine nodes. The set of correct nodes form a connected sub-mesh, and Byzantine nodes cannot eclipse correct nodes.*



Approach: *Providing each member with membership is a form of agreement. Previous works on Byzantine fault-tolerant agreement establish invariants that are impossible to invalidate. Even the most practical of these protocols require several rounds of all members broadcasting state to all other members, and can consequently not scale up to more than perhaps a few dozen members. In order to scale to thousands or more members, we had to come up with a radically different approach. Fireflies makes use of epidemic techniques ("gossip") to form a probabilistic agreement, which can only establish invariants that hold with a certain probability. Because invariants never hold for certain, defense against adversaries trying to break agreement can never rest.*

Accomplishments: *Fireflies has been adopted by several research projects around the world. For example, at UT Austin Prof. Alvisi and Dahlin are working to create scalable Byzantine and Rational fault-tolerant communication systems making use of game theory (incentives). While their work has been successful, they were not able to deal this far with dynamic systems in which members could come and go. They are now building on Fireflies to remedy this shortcoming of their work. In Norway, Prof. Johansen has developed a system, based on Fireflies, to dispatch security updates in a timely and coordinated manner. Traditional software updates are vulnerable to reverse engineering to discover software flaws. In Israel, Prof. Dolev is developing a self-stabilizing version of Fireflies in order to add additional immunity to attacks. At Cornell itself, Fireflies forms the basis for the SecureStream video streaming system (reported separately).*

Impact on the warfighter: Modern warfighter equipment will almost certainly carry devices that employ state-of-the-art distributed communication protocols. If one or more such devices were compromised, most such protocols could be easily attacked without the warfighter being able to tell the difference. As a result, a warfighter cannot put much trust in these devices. Fireflies is much less susceptible to such attacks and as a result allows warfighters to put significantly more trust in devices that use Fireflies.

References: (For downloading Fireflies, visit http://sourceforge.net/projects/fireflies)

Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays. Håvard Johansen, Andre Allavena, and Robbert van Renesse. Eurosys 2006. Leuven, Belgium. April 2006.

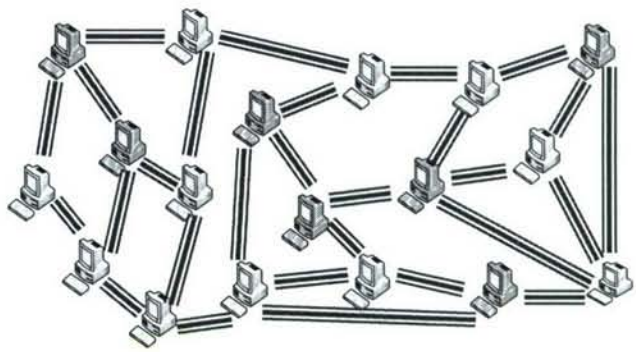
Correctness of Fireflies. Andre Allavena and Robbert van Renesse. Internal Report. June 2006.

FirePatch: Secure and Time-Critical Dissemination of Software Patches. Håvard Johansen, Dag Johansen, and Robbert van Renesse. IFIP International Information Security Conference (IFIPSEC 2007). Sandton, South-Africa. May 2007.

Self-Stabilizing and Byzantine-Tolerant Overlay Network. Danny Dolev and Robbert van Renesse. Submitted to OPODIS 2007. July 2007.

SecureStream: Intrusion-Tolerant Video Streaming

Goal: *Application-level multicast systems are vulnerable to attack that impede nodes from receiving desired data. Live streaming protocols are especially susceptible to packet loss induced by malicious behavior. We describe SecureStream, an application-level live streaming system built using a pull-based architecture that results in improved tolerance of malicious behavior. SecureStream is implemented as a layer running over Fireflies, an intrusion-tolerant membership protocol.*



Approach: *Our work introduces several techniques that reduce the opportunity for an attacker to compromise the quality of a streaming session, without incurring a high computational or network overhead. To repel forgery attacks, we employ an efficient packet authentication technique based on computing and distributing verification digests. To prevent attacks on the overlay structure (the membership protocol on top of which multicast systems operate), SecureStream is built upon Fireflies, a scalable one-hop Byzantine membership protocol.*

Accomplishments:

- *SecureStream is the first exploration of end-system attacks in the context of live streaming peer-to-peer protocols.*
- *We leverage previous work and present a comparison of different authentication protocols for signing and verifying packets efficiently in the context of application-level multicast.*
- *We thoroughly evaluate the effects of internal malicious peers on pull-based protocols. The issue is more serious than has previously been recognized.*
- *Aspects of our work are being adopted by other research groups. For example, at UT Austin Profs. Alvisi and Dahlin are using our linear digests. Prof. Porto at UFRGS in Brazil is extending SecureStream to deal with heterogeneity in the system.*

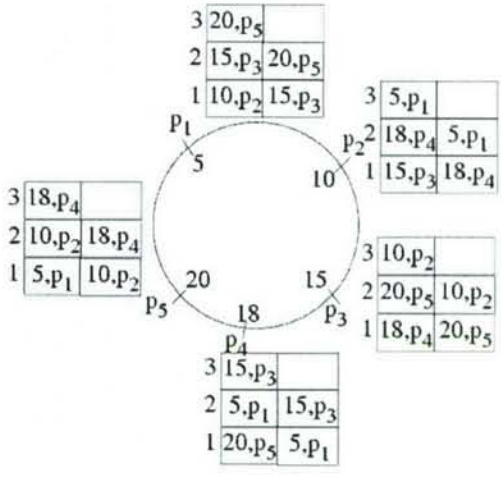
Impact on the warfighter: SecureStream can provide reliable delivery of streaming media to the warfighter even in the face of cyber-attacks.

References: (*Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*)

Maya Haridasan and Robbert van Renesse. *Defense Against Intrusion in a Live Streaming Multicast System*. 6th IEEE International Conference on Peer-to-Peer Computing (P2P2006). Cambridge, UK. September 2006.
Maya Haridasan and Robbert van Renesse. *SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination*. The Journal of Computer Communication's Special issue on Foundation of Peer-to-Peer Computing (accepted for publication). 2007.

P-RING: SCALABLE DISTRIBUTED RANGE QUERIES

Goal: *Modern data management systems are increasingly challenged by the need to support very large scale data sets distributed among many sites. The systems must allow distributed data to be queried in a manner similar to the ways centralized databases are queried. They must perform rapid dissemination of time-critical updates while providing strong consistency guarantees for concurrent queries, so that incoming data can reliably trigger a real-time response. Finally, the systems must be fault-tolerant, able to withstand high rates of churn with minimal effect on query performance or correctness.*



Approach:

We have developed P-Ring, a new peer-to-peer index structure that efficiently and scalably supports range queries as well as equality queries, and is robust even under high rates of churn. P-Ring can be viewed as an alternative approach to our earlier Kelips work. The Kelips design tolerates somewhat increased memory usage – O(sqrt(n)) – in exchange for O(1) file lookup times. This memory requirement is acceptable for current systems, but could eventually become a scalability bottleneck. P-Ring takes a different approach, using a hierarchy of fault-tolerant rings to provide O(log(n)) lookup time, by a protocol similar to a skiplist, with an improved memory requirement that is only polylog in n. The P-Ring also achieves excellent load balancing. A straightforward scheme yields a worst-case imbalance factor of 2, analogous to a B+ tree; while a more complicated but quite practical scheme can yield arbitrarily small imbalance factors with constant amortized overhead. A prototype implementation of P-Ring outperforms existing systems that attempt to provide similar guarantees.

Accomplishments:

Designed and implemented the P-Ring protocols and performed a thorough experimental evaluation.

- *Published the results in SIGMOD.*
- *Made our prototype implementation available as open source.*
- *Identified promising technology transition opportunities in collaboration with a local company (ATC-NY). Ongoing work is extending the system to a native XML database.*

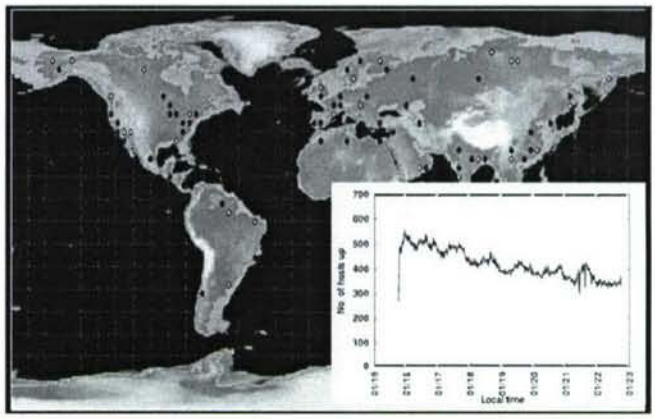
Impact on the warfighter: During military operations, timely and correct information can be vitally important. This work provides scalable correctness and responsiveness guarantees even under high rates of churn.

References:

P-Ring: An Efficient and Robust P2P Range Index Structure. Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke and Jayavel Shanmugasundaram. Proceedings 2007 ACM SIGMOD Conference. Beijing, China, June 2007.

Automated Availability Management

Goal: *A number of issues arise as Web Services and similar COTS components are migrated into GIG/NCES platforms. Our goal for developing automated availability management is to address the problem of providing highly-available services in distributed systems composed of relatively unavailable components. This problem arises in many settings, including ad-hoc wireless networks where disconnection and reconnection is frequent (e.g., battlefield scenarios), sensor networks (e.g., intelligence gathering), distributed computation and storage, etc.*



Approach: *A distinguishing feature across these disparate types of distributed systems is that they are composed of relatively unavailable components. Rather than being always available until failure, the components in these systems are both available and unavailable on a frequent basis (daily, even hourly), yet remain a functioning component of the system on long-term time scales (weeks to months to years). A fundamental challenge for designing and building next-generation distributed systems is providing high availability using these highly unavailable components. Automated availability management formalizes availability as an explicit property in distributed systems. Users and applications can request explicit availability guarantees for system objects and resources. For instance, in a wide-area distributed file system, users would specify that by default files require 99.9% availability over two years. To provide such guarantees, automated availability management uses (1) availability models to make efficient resource provisioning decisions and to predict and estimate future resource availability; (2) redundancy mechanisms to mask and tolerate component unavailability; and (3) repair policies to dynamically maintain resource availability in response to intermittent and permanent failure.*

Accomplishments: *We developed and evaluated a range of availability models, redundancy mechanisms, and repair policies across a spectrum of system configurations. As a concrete point in the system environment space, we measured the temporary and permanent failure characteristics of the Overnet peer-to-peer file sharing overlay network. This work was the first to study these characteristics in such systems, and the traces we gathered were used by many other research groups to evaluate their own efforts. We also refined automated availability management to exploit resource to improve system performance and reliability. Our goal has been to expose resource heterogeneity among nodes and tailor systems to take advantage of it.*

Impact on the warfighter: *Automated availability management provides a convenient abstraction for implementing highly available distributed services in situations where network disconnection and reconnection is frequent. The situations occur at many levels in military deployments, ranging from ad-hoc networks among troop patrols, battlefield communication networks linking troops, vehicles, and air support, carrier groups at sea, and the world-wide GIG IT infrastructure.*

References: (For downloads and complete list, visit http://sysnet.ucsd.edu/projects/recall/)

Replication Strategies for Highly Available Peer-to-Peer Storage Systems. Ranjita Bhagwan, David Moore, Stefan Savage, and Geoffrey M. Voelker. UCSD Technical Report No. CS2002-0726, November 2002.
Understanding Availability. Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Proceedings of the International Workshop on Peer To Peer Systems (IPTPS'03), Berkeley, CA, February, 2003.
Ranjita Bhagwan, Priya Mahadevan, George Varghese, and Geoffrey M. Voelker. Cone: A Distributed Heap-Based Approach to Resource Selection UCSD Technical Report CS2004-0784.
On Object Maintenance in Peer-to-Peer Systems Kiran Tati and Geoffrey M. Voelker. In Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS), Santa Barbara, California, February 2006.
Maximizing Data Locality in Distributed Systems, Fan Chung, Ron Graham, Ranjita Bhagwan, Geoffrey M. Voelker, and Stefan Savage, Journal of Computer and System Sciences 72(8), December 2006.

TotalRecall File System

Goal: *Our goal was to develop a specific system application of the automated availability approach. We designed a storage system called TotalRecall that applies automated availability management to large-scale, wide-area distributed storage systems. TotalRecall guarantees user-specified levels of data availability while minimizing the overhead needed to provide these guarantees in highly dynamic environments.*

Approach: *TotalRecall predicts the availability of its components over time, determines the appropriate level of redundancy to tolerate transient outages, and automatically initiates repair actions to meet user requirements. It uses replication and erasure coding to adapt the degree of redundancy and frequency of repair to the distribution of failures observed and predicted in the system. It also uses two repair strategies, eager repair and lazy repair, for trading off availability and replication overhead. Moreover, it closely approximates key system parameters, such as the appropriate level of redundancy, from a combination of underlying measurements and requirements. Finally, we have implemented and evaluated a prototype of TotalRecall that automatically adapts to changes in the underlying host population while effectively managing file availability and efficiently using bandwidth and storage resources.*

Accomplishments:

- *Developed and evaluated specific availability management mechanisms for storage systems, including availability models for short-term temporary and long-term permanent failures, erasure coding and mirroring redundancy mechanisms, and eager and lazy repair policies.*
- *Designed and implemented a prototype that runs on the PlanetLab testbed as the TotalRecall File System. Each participating host exports an NFSv3 file system interface to the system. External client hosts can mount TRFS and use it as any other remote NFS file system, storing data with high availability.*
- *Developed ``ShortCuts'', a routing approach for lookups that uses soft state to achieve routing performance that approaches the aggressive performance of one-hop schemes, yet uses an order of magnitude less communication overhead on average.*

Impact on the warfighter: *The TotalRecall File System provides a highly available distributed storage service in situations where network disconnection and reconnection is frequent. It is particularly useful in large-scale GIG IT infrastructures that face challenging communication constraints, such as among the many ships that comprise carrier groups operating at sea, as well as world-wide information infrastructure, such as the storage services supporting the data collection, analysis, and reporting performed by military branches and government agencies.*

References: (For downloads and complete list, visit http://sysnet.ucsd.edu/projects/recall/)

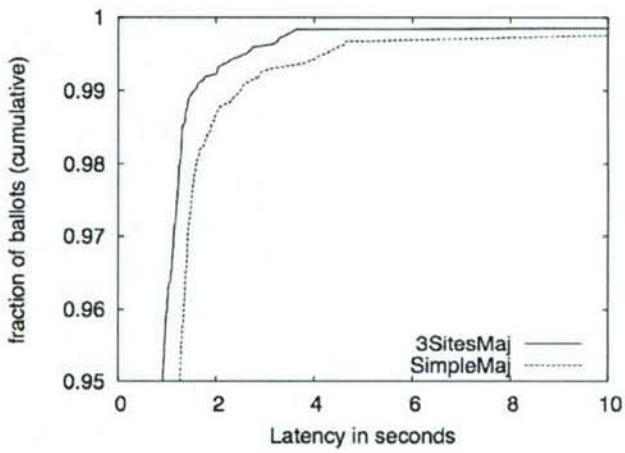
TotalRecall: System Support for Automated Availability Management Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. In Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, March 2004.

ShortCuts: Using Soft State To Improve DHT Routing. Kiran Tati and Geoffrey M. Voelker. In Proceedings of the Ninth International Workshop on Web Content Caching and Distribution (WCW'04), Beijing, China, October 2004.

Resource Reclamation in Distributed Hash Tables. Kiran Tati and Geoffrey M. Voelker. University of California, San Diego, CSE Technical Report CS2006-0863, July 2006.

Realistic Abstract Failure Models

Goal: *Failure models are part of the contract used in designing and deploying a distributed system. A failure model says what can go wrong with the environment, and so the system needs to be able to sustain its mission in the face of these adverse conditions. From a protocol design point of view, though, a failure model should be simple and abstract, since efficiency is obtained by leveraging off the details of the failure model. Practical details that are often ignored include non-identical failure rates, non-zero covariance of failures, and communications failures arising from BGP convergence issues.*



Approach: *One of the most fundamental protocols for fault tolerance is consensus, and so we deconstructed the various versions of this protocol to understand how it used the simple failure models for which it was developed. We identified two kinds of properties -* core *properties, useful describing failure scenarios, and* survivor set *properties, useful for showing that information is preserved despite failures. The two kinds of properties are duals of each other, and can be generalized to accommodate non-identical failures, non-zero covariance of failures, as well as many other failure patterns.*

Accomplishments: *We have generalized much of the lower bound results for consensus, quorum update, voting, and related problems. The results have been surprising: we have essentially developed a methodology for taking advantage of dependent failures. By knowing which failure patterns are more likely and which are not likely, replication of information can be done in an informed manner. In addition, we have found that some previous lower bounds are serendipitous: some difficulties with more general failure models appear only when dependent failures can occur. In addition, our work has shown that significant performance gains can be obtained using more accurate failure models. The graph above, for example, shows how a version of consensus has better availability and faster convergence time when the protocol makes use of the plausible failure patterns of a wide area network, even when no failures occur. Finally, we have developed a better understanding of the performance of core protocols in real*

Impact on the warfighter: *Failures of an information system in a real military deployment are complex and not easily characterized using the simple failure models commonly used in high-level protocol design before our work. Using the simpler models results in solutions that are slower and require more infrastructure, both of which pose problems in a deployment. With our models and new versions of basic protocols, it should be possible to build more efficient and robust information system.*

References:

Synchronous Consensus for Dependent Process Failures, Flavio Junqueira and Keith Marzullo, Proceedings of the International Conference on Distributed Computing Systems (ICDCS), Providence, Rhode Island, May 2003.

The Virtue of Dependent Failures in Multi-site Systems, Flavio Junqueira and Keith Marzullo, Proceedings of the IEEE Workshop on Hot Topics in System Dependability (HotDep), Yokohama, Japan, June 2005.

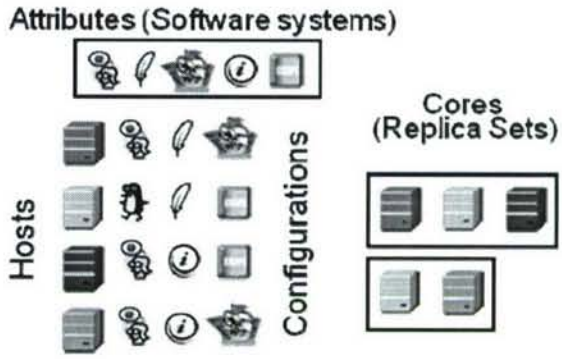
Replication predicates for dependent-failure algorithms, Flavio Junqueira and Keith Marzullo, Proceedings of the 11th International Conference on Parallel and Distributed Computing (EuroPar), August 2005.

Coterie Availability in Sites, Flavio Junqueira and Keith Marzullo, Proceedings of the International Symposium on Distributed Computing (DISC), Cracow, Poland, September 2005.

Classic Paxos vs. Fast Paxos: *Caveat Emptor*, Flavio Junqueira, Yanhua Mao and Keith Marzullo. Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07), Edinburgh, UK, June 2007.

Informed Replication

Goal: *Informed replication is an application of our dependent failure models to real distributed systems. Large-scale distributed systems are highly vulnerable to Internet catastrophes: events in which an exceptionally successful network pathogen, like a worm or email virus, causes data loss on a significant percentage of the machines connected to the network. Informed replication takes advantage of software heterogeneity among nodes in a system to efficiently and reliably ensure that replicated data and services survive.*



Approach: *The key observation that makes informed replication both feasible and practical is that Internet epidemics exploit shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, a pathogen that exploits one or more vulnerabilities cannot cause all replicas to fail. For example, to prevent a distributed system from failing due to a pathogen that exploits vulnerabilities in Web servers, the system can place replicas on hosts running different Web server software.*

Accomplishments:

- *Developed a system model using our core abstraction to represent failure correlation in distributed systems. A core is a reliable minimal subset of components such that the probability of having all hosts in a core failing is negligible.*
- *Measured and characterized the diversity of the operating systems and network services of hosts in the UCSD network to evaluate the degree of software heterogeneity found in an Internet setting.*
- *Developed heuristics for computing cores that provide excellent reliability guarantees, have low overhead, bound the number of replica copies stored by any host, and the heuristics lend themselves to a fully distributed implementation for scalability.*
- *To demonstrate the feasibility and utility of our approach, we applied informed replication to the design and implementation of the Phoenix cooperative, distributed remote backup system.*

Impact on the warfighter: *Informed replication provides a powerful approach for enabling large-scale distributed systems to survive virulent, self-propagating Internet malware. Systems that include hosts with different software configurations can take advantage of the approach, including systems deployed in the battlefield as well as military and agency IT infrastructure.*

References:

The Phoenix Recovery System: Rebuilding from the Ashes of an Internet Catastrophe, Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker, Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, HI, May 2003, pages 73-78.

Surviving Internet Catastrophes, Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker, Proceedings of the USENIX Annual Technical Conference, Anaheim, CA, April 2005, pages 45-60.

4. Personnel Supported

Faculty, Researchers, and PostDocs

Ken Birman (Cornell), Alan Demers (Cornell), Richard Eaton (Cornell), Paul Francis (Cornell), Johannes Gehrke (Cornell), Christoph Koch (Cornell), Keith Marzullo (UCSD), Jayavel Shanmugasundaram (Cornell), Niki Trigoni (Cornell), Robbert Van Renesse (Cornell), Mark Riedwald (Cornell), Geoffrey M. Voelker (UCSD), and Werner Vogels (Cornell), Walker White (Cornell).

Graduate Students

Jeanne Albrecht (UCSD), David S. Anderson (UCSD), Ben Atkin (Cornell), Alvin AuYoung (UCSD), Anand Balachandran (UCSD), Mahesh Balakrishnan (Cornell), Hitesh Ballani (Cornell), Rimon Barr (Cornell), Leeann Bent (UCSD), Ranjita Bhagwan (UCSD), Adrian Bozdog (Cornell), Cristian Bucila (Cornell), John Calandrino (Cornell), Zhiyuan Chen (Cornell), Sigmund Cherem (Cornell), Jessica Chiang (UCSD), Adina Crainiceanu (Cornell), Annemarie Dahm (UCSD), Abhinandan Das (Cornell), Chris Fleizach (UCSD), Flavio Junqueira (UCSD), Lin Guo (Cornell), Indranil Gupta (Cornell), Maya Hardisasan (Cornell), Mingsheng Hong (Cornell), Ken Hopkinson (Cornell), Tibor Janosi (Cornell), Prakesh Linga (Cornell), Ashwin Machanavajjala (Cornell), Priya Mahadevan (UCSD), Yanhua Mao (UCSD), Marvin McNett (UCSD), Alper Mizrak (UCSD), Krzysztof Ostrowski (Cornell), Biswanath Panda (Cornell), Ishwar Ramani (UCSD), Venugopala Ramasubramanian (Cornell), Manpreet Singh (Cornell), Michael Sirivianos (UCSD), Kiran Tati (UCSD), Ruijie Wang (Cornell), Ming Woo-Kawaguchi (UCSD), Dmitrii Zagorodnov (UCSD), Xianan Zhang (UCSD), Xinyang Zhang (Cornell).

Undergraduate Students

Justin Koser (Cornell), Ben Kraft (Cornell), Amar Phanishayee (Cornell).

Interactions/Transitions

The members of the team have been very active, speaking at a wide range of conferences and workshops during the course of the MURI project. In particular, Professor Birman maintains active dialog and research collaborations with many companies, and also advises the US military and government in many capacities. During the period of the MURI effort he has had active dialog with the following companies and US government agencies. Obviously, some of these have been more substantive than others, but as a group, they illustrate the extent of ties between Birman's effort and industry and support his view that as technologies emerge suitable for potential technology transition, there will be good opportunities for pursuing them further. For example, at the current time, his group is working to explore a transition path for the Ricochet technology into Raytheon's DDS platform for military publish-subscribe applications that need real-time responsiveness.

- Air Force Office of the CIO, Air Force Research Laboratories, Apache Group, Amazon, Cisco, DARPA, DHS, Microsoft, Infosys, IBM, Intel, Google, Mitre, NSF, OSD / DDR&E, OSD / DDS&T, RAND, Raytheon, SRI, White House OSTP, WSO2

Honors and Awards

Ken Birman is a Fellow of the ACM.

Johannes Gehrke received an Alfred P. Sloan Fellowship (2003).

Geoffrey M. Voelker received the UCSD Hellman Faculty Fellowship (2002), the UCSD Chancellor's Associates Award for Excellence in Undergraduate Teaching (2006), and the UCSD Jacobs School Ericsson Distinguished Scholarship (2007).

5. Key Research Publications

Finally, we include key research publications from each of the projects summarized above to provide depth and detail of our MURI research efforts.

Extensible Web Services Architecture for Notification in Large-Scale Systems

Krzysztof Ostrowski
Cornell University
krzys@cs.cornell.edu

Ken Birman
Cornell University
ken@cs.cornell.edu

Abstract

Existing web services notification and eventing standards are useful in many applications, but they have serious limitations precluding large-scale deployments: it is impossible to use IP multicast or for recipients to forward messages to others and scalable notification trees must be setup manually. We propose[1] a design free of such limitations that could serve as a basis for extending or complementing these standards. The approach emerges from our prior work on QSM [1]*, a new web services eventing platform that can scale to extremely large environments.*

1. Introduction

1.1. Motivation

Notification is a valuable, widely used primitive for designing distributed systems. The growing popularity of RSS feeds and similar technologies shows that this is also true at Internet scales. The WS-Notification [2] and WS-Eventing [3] standards have been offered as a basis for interoperation of heterogeneous systems deployed across the Internet. Unlike RSS, they are subscription-based, and hence free of the scalability issues of polling, and support proxy nodes that can be used to build scalable notification trees. Nonetheless, they embody restrictions:

- *Not self-organizing*. While both standards permit the construction of notification trees, such trees must be manually configured and require the use of dedicated infrastructure nodes ("proxies"). Automated setup of dissemination trees, formed by recipients, and without the dedicated infrastructure, is more appropriate.
- *Inability to use external multicast frameworks*. Both standards leave it entirely to the recipients to prepare their communication endpoints for message delivery. This makes it impossible for a group of recipients to dynamically agree upon a shared IP multicast address, or to construct an overlay multicast within a segment of the network. Yet such techniques are central to achieving high performance and scalability, and could also be used to provide QoS guarantees or leverage the emergent technologies.
- *No forwarding among recipients*. Many content distribution schemes build overlays within which content recipients participate in message delivery. In current web services notification standards, however, recipients are *passive* (limited to data reception).
- *Difficult to manage*. At Internet scales, it is hard to create and maintain a dissemination structure that would permit any node to serve as a publisher or subscriber, for this requires many parties to maintain common infrastructure, agree on standards, topology and other factors. Any such large-scale infrastructure should respect local autonomy, whereby the owner of a portion of a network can set up policies for local routing, availability of IP multicast, etc.
- *Weak reliability*. Reliability in the existing schemes is limited to per-link guarantees, resulting from the use of TCP. In many situations, stronger guarantees are required, e.g. to support virtually synchronous, transactional or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages or participate in multi-party protocols, even weak guarantees cannot be provided.

1.2. Our Contribution

In this document, we propose a principled approach to building large-scale systems for web services notification. We outline a design for an extensible notification scheme free of the limitations just described, which is the basis for Quicksilver [1], a new scalable and reliable publish-subscribe and notification platform under development at Cornell. Motivated by the end-to-end principle, we separate the implementation of loss recovery and strong reliability properties from the unreliable dissemination of messages. Accordingly, our design includes a *reliability framework* and a *dissemination framework*: two largely independent, yet complementary structures.

Both frameworks reflect the principles articulated below, and they share many elements. In particular, both employ hierarchical protocol stacks, an idea that is central to our architecture. These stacks permit the definition of an Internet-scale loss recovery scheme which can employ different recovery policies within different administrative domains. Likewise, they permit a construction of a global dissemination scheme that uses different mechanisms to distribute data within different administrative domains.

[1] Our effort is supported by AFRL/Cornell Information Assurance Institute.

1.3. Design Principles

The limitations of the existing designs listed above and our experience designing scalable multicast systems led us to the following design principles:

- *Programmable nodes*. Senders and recipients should not be limited to sending or receiving. They should be able to perform certain basic operations on data streams, such as forwarding or annotating data with information to be used by other peers, in support of local *forwarding policies*. The latter must be expressive enough to support protocols used in today's content delivery networks, such as overlay trees, rings, mesh structures, gossip, link multiplexing, or delivery along redundant paths.
- *External control*. Forwarding policies used by nodes must be selected and updated in a consistent manner. A node cannot predict a-priori what policy to use, or which other nodes to peer with; it must permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering etc.
- *Hierarchical structure*. The principles listed above should apply to not just individual nodes, but also entire administrative domains such as LANs, data centers or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating the cooperation among organizations in maintaining the global infrastructure. The way a message is delivered to subscribers across the Internet thus reflects policies defined at various levels.
- *Isolation and local autonomy*. A certain degree of local autonomy of the administrative domains must be preserved; such as how messages are forwarded internally, which nodes create which communication endpoints etc. In essence, the structure of a domain should be hidden from other domains it is peering with and from the higher layers. Likewise, details of its own subcomponents should as opaque as possible.
- *Channel negotiation*. Communication channel creation should permit a handshake. A recipient might be asked to e.g. join an IP multicast group, or subscribe to an external system. The recipient could then make a configuration decision on the basis of the information about the sender, e.g. a LAN asked to prepare a communication endpoint for receiving may choose a well-provisioned node to handle the anticipated load.
- *Managed channels*. Communication channels should be represented as *active contracts* in which receivers have a degree of control over the way the senders are sending. In self-organizing systems, reconfiguration triggered by churn is common and communication channels often need to be reopened or updated to adapt to the changing topology, traffic patterns or capacities. For example, a channel that previously requested that a given source transmits messages to one node, might notify the source that messages should now be transmitted to two other nodes, instead.
- *Reusability*. It should be possible to specify a policy for message forwarding or loss recovery in a standard way and post it into an online library of such policies as a contribution to the community. Administrators willing to deploy a given policy within their administrative domain should be able to do so in a simple way, e.g. by drag-and-drop, within a suitable GUI.

1.4. Basic Concepts

We employ the usual terminology, where notifications are associated with *topics* and produced by *publishers* and delivered to *subscribers*. We use the term "*group X*" to refer to the group of nodes subscribed to topic "*X*". More than one publisher may exist for a given topic. The prospective publishers and subscribers register with a *subscription manager*, which can be decentralized and independent of the publishers (see Figure 1, top). In our architecture, nodes reside in *administrative domains*. Nodes in the same domain are jointly managed. It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries; either for administrative reasons, or because communication locally in a domain is cheaper than across domains, as it is often related to network topology. Publishers and subscribers may be scattered across organizations, which must cooperate in message delivery. This often presents a logistic challenge (see Figure 1, bottom).

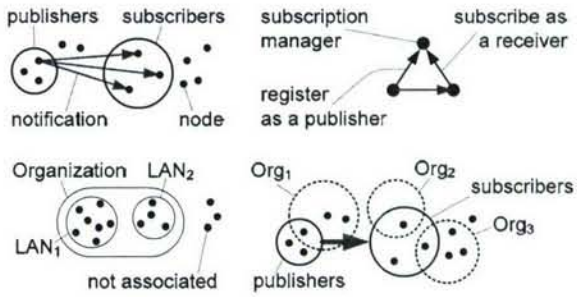


Figure 1. Publishers and subscribers register with the *subscription manager* (top). Nodes are scattered across *administrative domains* hierarchically divided into *sub-domains* (bottom).

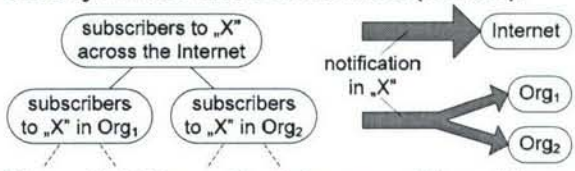


Figure 2. A hierarchical decomposition of the set of subscribers along the domain boundaries.

1.5. A Hierarchical View of the Network

A group X of subscribers for a given topic across the Internet can be divided into subsets Y_1, Y_2, ..., Y_N of subscribers in N top-level administrative domains (Figure 2). This can be continued recursively, leading to a hierarchical perspective on the set of all subscribers. By the principle of *isolation* and *local autonomy*, each administrative domain should manage the registration of its own publishers and subscribers internally, and decide how to distribute messages among them according to its local policy. Similar ideas were previously exploited in the context of content-based filtering [6], and in many scalable multicast algorithms, e.g. in RMTP [4]. This also reflects the principle of locality, implicit in many scalable protocols. Following this principle, groups of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection etc. Each such group is treated as a single cell within a global infrastructure. A protocol running at a global level connects all cells into a single structure. Scalability arises as in the *divide and conquer* principle. Additionally, the cells can locally share workload and amortize overhead, e.g. buffer messages coming from different sources and locally disseminate such combined bundles etc. We make heavy use of this property in our QSM [1] system.

This principle of locality and the hierarchical view of the network outlined above form the basis for our design.

2. Design Overview

2.1. The Hierarchy of Scopes

Our design is constructed upon the following principal concepts: *management scope*, *channel*, *filter*, *forwarding policy*, *session*, *recovery algorithm*, and *recovery domain*.

A *management scope (*or simply a *scope)* represents a *set of jointly managed nodes*. It may include a single node, span over a group of nodes residing within a certain administrative domain, or include nodes clustered based on other criteria, such as common interest. In the extreme, a scope may span the Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but it will often be the case, and we will refer to a *LAN scope* (or just a *LAN*) to mean the scope spanning over all nodes residing within a LAN. The reader might find it easier to understand our design with such examples in mind.

A scope is not just any group of nodes, the assumption that they are *jointly managed* is essential. The existence of a scope is dependent upon the existence of infrastructure that maintains its membership and administers it. For a scope that corresponds to a LAN, this could be a server managing all local nodes. In a domain that spans several data centers in an organization, it could be a management infrastructure with a server in the company headquarters indirectly managing the network via subordinate servers in data centers. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the global scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many independently managed global scopes could thus co-exist.

Like administrative domains, scopes form a hierarchy, defined by a relation of *membership*: a scope may *declare* itself to be a *member* (*sub-scope*) of another scope. If X declares itself to be a member of Y, it means X is either physically or logically a part (or subset) of Y. Typically a scope defined for a sub-domain X of some administrative domain Y will be a member of the scope defined for Y. For instance, a node would be a member of a LAN. The LAN would be a member of a data center, which in turn would be a member of a corporate network etc. A node could also be a member of a scope of some overlay network. For a data center, two scopes might be defined, e.g. a monitoring scope and a control scope, both scopes covering the entire data center, with some LANs being a part of one scope, the other scope, or both. The corporate scope could be a member of several Internet-wide scopes.

The generality in these definitions allows us to model various special cases, such as clustering of nodes based on interest or other factors. Such clusters, formed e.g. by a server managing a LAN and based on node subscription patterns, could also be considered scopes, all managed by the same server. Nodes would be members of clusters and clusters (not nodes) would be members of the LAN. As it will be explained below, each cluster, as a separate scope, could be locally and independently managed. In [1], such construction is a basis for our scalable multicast protocol.

A scope hierarchy is not a tree. There may be multiple global scopes, or many super-scopes for any given scope. However, a scope always decomposes into a tree of subscopes, down to the level of nodes. We refer to a *span of a scope X* as the set of all nodes at the bottom of a hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, i.e. such that all subscribers to X reside in the span of X. Publishing a message in a topic is thus equivalent to delivering it to all subscribers in the span of some global scope, which may be further decomposed into subscribers in the spans of sub-scopes (compare section 1.5 and Figure 2).

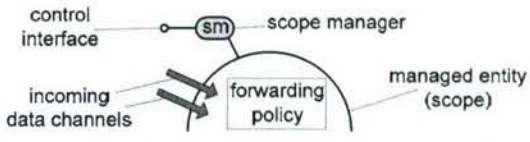


Figure 3. Accessed via a control interface and configured with a forwarding policy, a scope manager creates incoming data channels.

2.2. The Anatomy of a Scope

The infrastructure managing a scope is referred to as a *scope manager* (SM). A single SM may control multiple scopes. It may be hosted on a single node, or on a set of nodes, perhaps outside of the scope it controls. It exposes a *control interface*, a web service hosted at a well-known address, to dispatch control requests directed to scopes it controls (Figure 3). SMs interact by calling each other's control web interfaces (see also [8]).

A scope maintains communication *channels* for use by other scopes. A *channel* is a mechanism through which a message can be delivered to all those nodes in the span of this scope that subscribed to any of a certain set of topics. In a scope spanning over a single node, a channel may be just an address/protocol pair; creating it would mean arranging for a local process to open a socket. In a distributed scope, a channel could be an IP multicast address; creating it would require all local nodes to listen at this address. In an overlay network, a channel could lead to nodes that forward messages across the entire overlay.

A scope that spans over a set of nodes is governed by forwarding *policy* specifying how messages that originate within that scope or arrive through some communication channel are forwarded internally and to other scopes.

The reader will recognize in our construction the principles we formulated earlier. Scopes, whether individual nodes, LANs or overlays, are *externally controlled* using their control interfaces, may be *programmed* with policies that govern the way messages are distributed internally and forwarded to other scopes, and transmit messages via *managed* communication channels established through a dialogue between a pair of SMs.

2.3. Hierarchical Composition of Policies

Following our design principles, we propose to solve the issue of a large-scale global cooperation in message delivery between independently managed administrative domains by introducing a hierarchical structure, in which forwarding policies defined at various levels are merged into a single dissemination scheme. Each scope is configured with a policy dictating, on a per-topic (and perhaps a per-sender) basis, how messages are forwarded among its members. For example, a policy governing a global scope might determine how messages in topic T, originating in a corporate network X, are forwarded between the various organizations. A policy of a scope of the given organization's network might determine how to forward messages among its data centers, and so on. A policy defined for a particular scope X is always defined at the granularity of X's members (not individual nodes). The way a given sub-scope Y of X delivers messages internally is a decision made by Y autonomously. Similarly, X's policy may specify that Y should forward messages to Z, but it is up to Y's policy to determine how to perform this task.

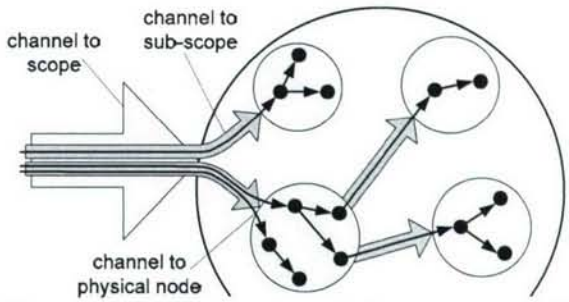


Figure 4. Channels created in support of forwarding policies defined at different levels.

Figure 5. Forwarding graphs for different topics are superimposed. Two members may be linked by multiple channels, each with a different filter.

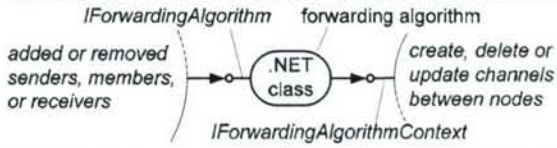


Figure 6. A forwarding policy as a code snippet.

Accordingly, a global policy may request organization X to forward messages in topic T to organizations Y and Z. A policy governing X may then determine that to distribute messages in X, they must be sent to LAN_1, which will forward them to LAN_2. The same policy might also specify which LANs within X should forward to Y and Z. Finally, the policies of the respective LANs could delegate these tasks to individual nodes. When the policies defined at all the scopes involved are combined, the resulting *forwarding structure* completely determines the way messages are forwarded (see Figure 4).

In the examples above, the policies are simply graphs of connections: each message is always forwarded along every channel. In general, however, each channel may be optionally constrained by a *filter* that decides, on a per-message basis, whether to forward or not, and optionally tags the message with custom attributes. This allows us to express many popular techniques, e.g. using redundant paths, multiplexing between dissemination trees etc.

Every scope manager maintains internally a mapping from topics to policies: a graph of channels to create and filters on them. Such graphs of connections for different topics are superimposed (see Figure 5). Based on this, the SM asks the scope members to create channels and filters. When the structure is modified as a result of membership or subscription changes, the SM makes additional control requests to reflect this.

In our framework, a policy is defined as an algorithm that lives in an *abstract context*, with a fixed set of *events*

to respond to, standard set of *operations* and *attributes* to inspect. In a prototype we are developing, we implement a forwarding policy as a .NET class, stored in a DLL on an *algorithm library* server, that implements an abstract interface and interacts with an abstract *context* hiding the details of the environment (Figure 6). This allows our policies to be used within any scope.

2.4. Communication Channels

Consider a node X, a member of a scope Z that, based on a forwarding policy at Z, has been requested to establish a communication channel to scope Y to forward messages in topic T. Following the protocol, X asks the SM of Y for the specification of the channel to Y that should be used for messages in topic T. The SM of Y might respond with an address/protocol pair that X should use to send over this channel. Alternatively, a forwarding policy defined for T at scope Y may dictate that, in order to send to Y in topic T, X should establish channels to members A and B of Y, constrained with filters α and β. After X learns this from the SM of Y, it contacts SMs of A and B for details. Notice how the channel decomposes into sub-channels to A and B through a policy at a target scope Y.

This decomposition continues hierarchically, until the point when scope X is left with a tree containing filters in internal nodes and address/protocol pairs at the leaves (Figure 9). In order to send a message along the channel built in this way, X executes filters to determine which sub-channels to use, proceeding recursively, until it is left with a list of address/protocol pairs, then transmits the message. Filters will typically be simple, such as modulo-**n**; hence X could perform this procedure very efficiently.

Accordingly, to support the hierarchical composition of policies described in the preceding section, we define a channel as one of the following: an address/protocol pair, a reference to an external multicast mechanism, or a set of sub-channels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which sub-channels to use for sending, on a per-message basis (see Figure 7 and Figure 8).

Consider now the case when scope X, spanning over a set of nodes, has been requested to create a channel to scope Y. Through a dialogue with Y and its sub-scopes, X can get a detailed channel definition, but unlike in the example above, X now spans over a set of nodes, and as such, it cannot *execute* filters or *send* messages.

We propose two example generic techniques that solve this problem: *delegation* and *replication* (Figure 10). Both rely on the fact that if scope X receives messages in a topic T, then some of its members, Z, must receive them (for otherwise X would not be made part of a forwarding structure for topic T by X's super-scope). In case of *delegation*, X requests such a sub-scope Z to create the channel on behalf of X, essentially delegating the whole channel. The problem can be recursively delegated, down to the level where a single physical node is requested to create a channel. A more sophisticated use of delegation would be for X to delegate sub-channels. In such case, X would first contact Y to obtain the list of sub-channels and the corresponding filters, and for each of these sub-channels, delegate it to one of its sub-scopes. In any case, X delegates the responsibility for sending over a channel, in one way or another, to one or more of its sub-scopes.

In case of *replication*, scope X requests **n** of its sub-scopes to create the channel, but constrains each with a modulo-**n** filter based on a message sequence number (i.e. sub-scope **k** only forwards messages with numbers **m** such that **m** mod **n** equals **k**), effectively implementing a round-robin policy. Although all sub-scopes would create the same channel, the round-robin filtering policy ensures that every message is forwarded only by one of them.

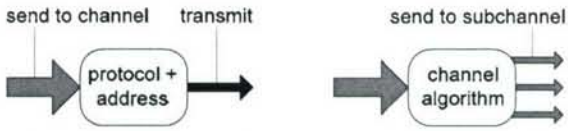


Figure 7. A channel may be an *address/protocol* pair (left), or it may consist of *sub-channels*, with an *algorithm* deciding what goes where (right).

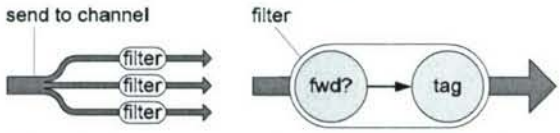


Figure 8. Channel algorithms are realized as sets of filters, one per subchannel, deciding whether to forward, and optionally adding custom tags.

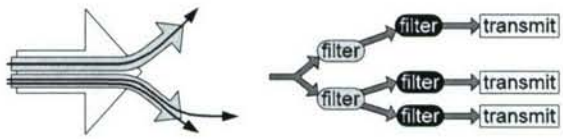


Figure 9. A channel split into sub-channels and a possible filter tree corresponding to it.

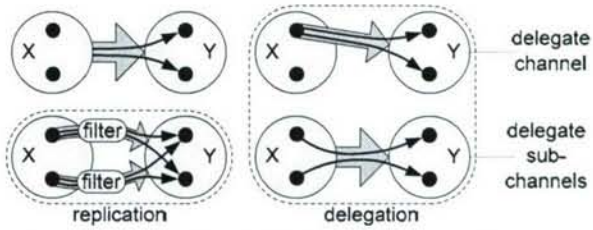


Figure 10. A distributed scope may delegate a channel or its sub-channels to members, or it may replicate them among members with filters that jointly implement a round-robin policy.

2.5. Reliability Scopes

The design of the reliability framework also relies on the concept of management scopes, referred to here as *reliability scopes* (in contrast to *dissemination scopes* in the dissemination framework). A reliability scope isolates and encapsulates the local aspects of loss recovery, hiding details from other scopes, just like a dissemination scope hides the local aspects of message delivery. Reliability scopes are also controlled by scope managers. Both kinds of scopes would typically overlap. For example, a single scope could be defined for an administrative domain such as a LAN, isolating local aspect of both dissemination and reliability. The scope could then be controlled by a single SM managing both dissemination and reliability.

The separation of dissemination from reliability makes it possible to combine an arbitrary unreliable notification mechanism, such as IP multicast or an overlay content delivery system, with a wide range of reliability protocols expressible in our reliability framework. This degree of reusability has not been possible with prior architectures.

2.6. Hierarchical Approach to Reliability

Our approach to reliability resembles our approach to dissemination. Just as channels are decomposed into sub-channels, in the reliability framework we decompose the task of repairing after message losses and providing other reliability goals. Recovering messages in a certain scope is modeled as recovering within sub-scopes, and then recovering "among" the sub-scopes (Figure 11). Just like recovery among single nodes, recovery among LANs may involve comparing their "state" (such as aggregated ACK or NAK information for the entire LANs) and forwarding lost messages. In section 2.9 we give examples of how recovery protocols may be defined and combined.

In our framework, different recovery schemes may be used in different scopes, to reflect differences in network topology, node or network capacity, the way subscribers are distributed (e.g. clustered vs. scattered around) etc.

Just like messages are disseminated through channels, reliability is achieved via *recovery domains*. A recovery domain **D** in scope **X** may be thought of as a "distributed recovery protocol running among some nodes in **X**" that performs recovery-related tasks for a certain set of topics. The concept of a recovery domain is symmetric, dual to the notion of a channel. We present it via analogy.

- Just like a channel is created to disseminate messages for some topics $\mathbf{T_1}$, $\mathbf{T_2}$, ... $\mathbf{T_k}$ in scope **X**, a recovery domain is created to handle loss recovery and other reliability tasks, again for a specific set of topics and in a specific scope. Just like there may be multiple channels to a scope, e.g. for different sets of topics, multiple recovery domains, each for different topics, may exist within a single reliability scope.
- Just like channels may be composed of sub-channels, a recovery domain **D** defined at scope **X** may be composed of *sub-domains* $\mathbf{D_1}$, $\mathbf{D_2}$, ... $\mathbf{D_n}$ defined at sub-scopes of **X** (we will call them *members* of D). Each such sub-domain $\mathbf{D_i}$ handles recovery for a certain set of subscribers in the respective sub-scope, while **D** handles recovery "across" its sub-domains.
- Just like channels are composed of sub-channels via applying filters assigned by forwarding policies, a recovery domain **D** performs its recovery tasks using a *recovery algorithm*. Such an algorithm, assigned to **D**, specifies how to combine recovery mechanisms in the sub-domains of **D** into a mechanism for all of **D**. Recovery algorithms are defined in terms of how the sub-domains "interact" with each other. We will see how this is achieved in section 2.9.
- Just like a single channel may be used to disseminate messages in multiple topics, a recovery domain may run a single protocol to perform recovery for multiple topics. In both cases, reusing a single mechanism (a channel, a token ring etc.) may significantly improve performance due to the reduction in the total number of "control" messages. We evaluated this idea in [1].

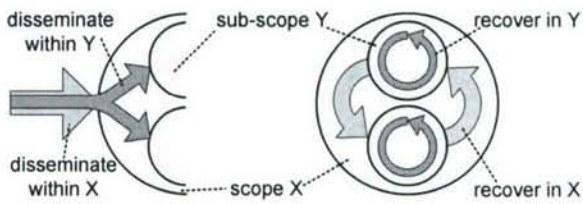


Figure 11. The similarity between hierarchical dissemination (left) and recovery (right).

Each individual node is a recovery domain on its own. In a distributed scope such as a LAN, on the other hand, two cases are possible. First, a single domain may cover the entire LAN. All internal nodes could form e.g. a token ring, exchange ACKs for messages in all topics, and use this to arrange for local repairs. Another possibility is that separate domains would be created for every individual topic. Subscribers to different topics would form separate structures, such as ring or trees, and run separate protocol instances in each, exchanging state and los messages.

As explained later, recovery domains in our system actually handle recovery for specific *sessions*, not just for specific topics. Sessions are introduced in section 2.7.

A recovery domain D of a data center could have as its members recovery domains created in LANs. Note that in this case, members of D would be sets of nodes. A recovery algorithm running in D would specify how all these different sets of nodes should exchange state and forward lost messages to one another. Note the similarity to a forwarding policy in a data center, which would also specify

how messages are forwarded among sets of nodes. As shown in section 2.10, recovery algorithms are implemented through delegation, just like forwarding. A concept of a *recovery algorithm* is, to a certain extent, symmetric to the notion of a forwarding policy.

2.7. Sessions

Within our architecture, protocols that provide strong reliability guarantees express them in terms of *epochs*. An epoch corresponds to what in group communication literature is called a *membership view*. The lifetime of a topic is divided into a sequence of epochs. Whenever the set of subscribers to a topic changes as a result of a subscribe/unsubscribe request or a failure, the event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs. One then defines consistency in terms of which messages may be delivered to which subscribers and at what time, relative to epoch boundaries. The traditional term "membership view" reflects the fact that epochs begin and end with membership change events. The set of subscribers during a given epoch is fixed.

Although simple protocols, such as SRM or RMTP, do not rely on a consistent view of group membership, and their properties are not defined in terms of epochs, epochs are still a useful, if not a universal concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, e.g. to rebuild a distributed structure, release resources or cancel activity that is no longer necessary. Many simple protocols simply do not take this factor into account.

We introduce the idea of a *session*, a generalization of an epoch (membership view). A session is also an epoch in the prior sense, i.e. the lifetime of any given topic can always be divided into a sequence of sessions. Like before, any membership change marks the beginning of a new session and for a given session, membership is fixed. However, a new session may also be initiated even if membership is unchanged. The reliability properties of a group may vary to some extent in the subsequent sessions. An important example is an administrative change, where a new protocol is introduced, e.g. because it is more efficient or to fix a bug in the existing protocol. In Internet-scale systems such administrative changes must be performed online; session changes achieve this.

Session numbers are assigned globally for consistency. As explained before, for a given topic, a single "global" scope always exists such that all subscribers to that topic reside within the span of this scope. This is true for both dissemination and reliability frameworks. Usually, both global scopes overlap and are managed by a single SM. The top-level SM assigns and updates session numbers. Note that local topics (e.g. internal to an organization) could be managed by the local SM, much in a way local newsgroups are visible and managed locally.

Before discussing the mechanisms used to manage membership, we conclude the discussion of sessions by explaining how they impact the behavior of publishers and subscribers. After registering, a publisher waits for the SM to notify it of the session number to use for a particular topic. A publisher is also notified of changes to the session number for topics it registered with. All published messages are tagged with the most recent session number, so that whenever a new session is started for a topic, within a short period of time no further messages will be sent in the previous session. Old sessions eventually quiesce as receivers deliver messages and the system completes flushing, cleanup and other reliability mechanisms used by the particular protocol. Similarly, after subscribing to a topic, a node does not process messages tagged as committed to session **k** until it is explicitly notified that it should receive messages in that session. Later, after session **k+1** starts, all subscribers are notified that session **k** is entering a *flushing* phase (this term originates in *virtual synchrony* protocols, but similar mechanisms are common in reliable protocols; a protocol lacking a flush mechanism simply ignores such notifications). Eventually, subscribers report that they have completed flushing and a global decision is made to cease activity and *cleanup* resources pertaining to session **k**, completing the transition.

2.8. Constructing the Recovery Structure

Reliable protocols often rely on, or could benefit from, a consistent view of membership. It helps to determine which nodes have crashed or disconnected. In existing systems, this is achieved by a Global Membership Service (GMS) that monitors failures and membership changes, decides when to install new membership views for topics, and notifies all affected members of the new views. In our framework, the global SM for a given topic is responsible for announcing when sessions begin and end. However, if the global SM had to process all subscriptions, it would lead to a non-scalable design that violates the principle of isolation. To avoid this, for each topic **T** we distribute the information about membership of **T** across all SMs in the hierarchy of scopes for **T** (this hierarchy was defined in section 2.1). Each SM thus has only a partial membership view for each session. This scheme is outlined below.

In the reliability framework, if a scope **X** subscribes to a topic **T**, it specifies some local recovery domain **D** that should handle the recovery for topic **T** in **X**. The **X**'s super-scope **Y** processes this subscription request jointly with requests from other sub-scopes. It then creates its own recovery domains, with the newly subscribed and perhaps some existing sub-domains as members, and then issues its own subscription requests to its super-scope. This continues recursively up to the global scope.

The scheme used by the super-scope to create recovery domains must abide by three rules. First, the list of sub-

domains of a recovery domain is determined once at the time of creation, and fixed throughout its lifetime. This is necessary to ensure that a hierarchical structure employed for recovery in any given session does not change, which simplifies the overall design. Second, a recovery domain **D** at scope **X** is responsible for handling recovery for a specific set of topics, in specific sessions. If a change in membership in any of these topics occurs locally in **X**, a new recovery domain **D'** must be created, and when a new session is announced, it is installed in **D'**. This is because the existing recovery domain **D** no longer represents the current set of subscribers within **X**, hence a new distributed structure **D'** must be established. Finally, if a new session is announced for some topic **T**, but no membership changes occurred for **T** within scope **X** since the previous session, then an existing recovery domain should be re-used to handle recovery in the new session.

In a scope in which recovery for each topic is handled individually, we would maintain a separate sequence of recovery domains for each topic. A new domain would be created whenever the set of subscribers locally changes. In a scope in which recovery for all topic is performed jointly, such as e.g. in a cluster of nodes defined based on subscription patterns in which all nodes are subscribers to the same set of topics, there will be just a single sequence of recovery domains. We used the latter scheme in [1].

The above procedure effectively constructs a hierarchy of sub-domains, with the property that for each topic **T**, the recovery domains subscribed to **T** form a tree.

The global scope assigns new session numbers for all topics for which subscribe or unsubscribe requests have been received, and determines which of its local recovery domains should handle the new sessions. This represents a coarse-grained membership view, for each session only top-level recovery domains are specified, with no further details. The information about the new sessions is now sent down the tree of subscribers, and transformed along the way to filter out unnecessary details. The membership information a scope **X** receives for a session **S** is limited to one level "above" **X**, i.e. it includes **X**'s own recovery domain that got subscribed to **S** and the recovery domains of its *sibling* scopes (i.e. scopes that have the same super-scope). It is also coarse-grained, i.e. it does not provide any details at the level "below" **X** or its siblings.

2.9. Modeling Recovery Algorithms

The design of the reliability framework is based on an abstract model of a distributed protocol dealing with loss recovery and other reliability properties. When expressed within our framework, such protocols will be referred to as *recovery algorithms*. Recovery algorithms are the basic building blocks in constructing our hierarchical reliability protocols, much in a way channels and filters are the basic building blocks in our forwarding infrastructure.

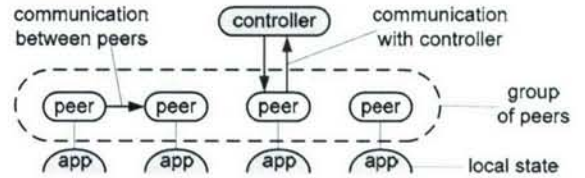


Figure 12. A group of *peers* in a reliable protocol.

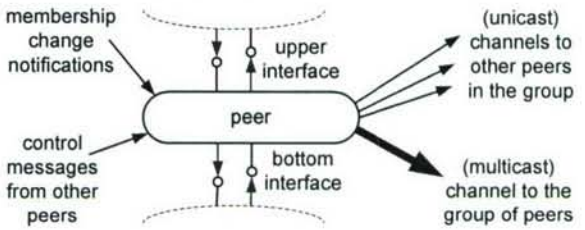


Figure 13. A peer modeled as a component living in abstract environment (events, interfaces etc.).

A protocol such as SRM, RMTP, or virtual synchrony is defined in terms of a group of cooperating *peers* that send control messages and forward lost packets to each other, and perhaps to a distinguished node, such as a sender or some node higher in a hierarchy, that we will refer to as a *controller* (Figure 12). The controller does not have to be a separate node; this function could be served by one of the peers. The distinction between the peers and the controller may be purely functional. The point is that the group of peers, as a whole, may be asked to perform a certain action, or calculate a value, for some higher-level entity, e.g. a sender, a higher-level protocol, or a layer in a hierarchical structure etc. Examples of such actions include requesting or performing a retransmission for all nodes, reporting which messages were successfully delivered to all nodes etc. Irrespectively of how exactly the interaction with a controller is realized, it is present in this form or another in almost every protocol run by a set of receivers. We shall refer to it as an *upper interface*.

Each peer inspects and controls *local state*. Such state may include e.g. a list of messages received and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered etc. Operations a peer may issue to change the local state could include e.g. retrieving/purging messages from a local cache, marking messages as "deliverable", handing a previously missed message to the application or assigning message sequence in a "totally ordered" group. We refer to such operations, used to view or control local state, as a *bottom interface*.

In protocols offering strong guarantees, peers typically know the membership of their group, received as a part of the initialization process, and subsequently updated via *membership change* events. Peers send control messages to each other to share state or to request actions, such as forwarding messages. Sometimes, as in SRM, a multicast channel to the entire peer group exists.

To summarize, in most reliable protocols a peer can be modeled as running in an environment that provides the following: a *membership view* of its peer group, *channels* to all other peers, and sometimes to the entire group, a *bottom interface* to inspect or control local state, and an *upper interface* to interact with a sender or a higher level in the hierarchy concerning the state of the whole group, (Figure 13). In some protocols, parts of the environment might be unavailable, e.g. in SRM peers might not know other peers. The bottom and upper interfaces would vary.

This model is flexible enough to capture the key ideas and features of a wide class of protocols, including virtual synchrony. However, because in our framework protocols must be reusable in different scopes, they may need to be expressed in a slightly different way, as explained below.

In the RMTP protocol [4], the sender and the receivers for a given topic form a tree. Within this tree, each subset of nodes consisting of a parent and child nodes serves as a separate, local recovery group. The child nodes in every such group send their local ACK/NAK information to the parent node, which arranges for a local recovery within the recovery group. The parent itself is either a child node in another recovery group, or it is a sender, at the root of the tree. Packet losses in this scheme are recovered on a hop-by-hop basis, top-down or bottom-up, one level at a time. This scheme distributes the burden of processing the individual ACKs/NAKs, and of retransmissions, which is normally the responsibility of the sender. This improves scalability and prevents the "ACK implosion".

There are two ways to express RMTP in our model. One approach is to view each recovery group consisting of a parent node and its child nodes as a separate group of peers (Figure 14). Since internal nodes in the RMTP tree simultaneously play two roles, a "parent" node in one recovery group and a "child" node in another, we think of a node as running two "agents", each representing a different "half" of the node and serving as a peer in a separate peer group. Every group of peers, in this perspective, includes the "bottom agent" of a parent node and "upper agents" of child nodes. When a node sends messages to its child nodes as a result of receiving a message from its parent, of vice versa, we may think of those two "agents" as interacting with each other through a certain interface that one of them views as upper, and the other as bottom. These two agents play different roles, as explained below.

The bottom agent of each node interacts via its *bottom interface* with the local state of the node. It also serves as a distinguished peer in the peer group composed of itself and the upper agents of child nodes. A protocol running in this peer group is used to exchange ACKs between child nodes and the parent node and arrange for message forwarding between peers, but also to calculate collective ACKs for the peer group, i.e. which messages were not recoverable in the group. This is communicated by the bottom agent, via its *upper interface*, to the upper agent.

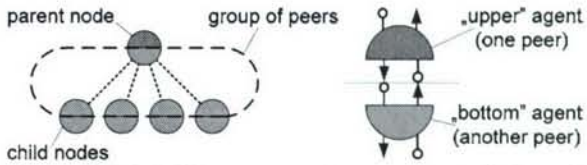


Figure 14. RMTP expressed in our model. A node hosts "agents" playing different roles.

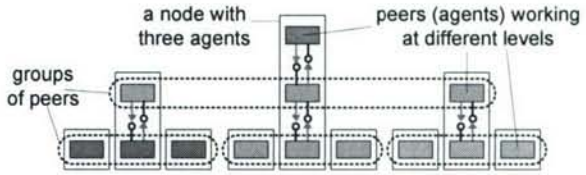


Figure 15. Another way to express RMTP. Each node hosts multiple "agents" that act as peers at different levels of the RMTP hierarchy.

The upper agent of every node interacts via its *bottom interface* with the bottom agent. What the upper agent considers as its "local state" is not the local state of the node. Instead, it is the state of the entire recovery group, including the parent and child nodes, that is collected for the upper agent by the bottom agent.

Such interactions, between a component that is a part of a "higher layer" and a component that resides in a "lower layer", both components co-located on the same physical node and connected via their upper and bottom interfaces, are the key element in our architecture.

At the top of this hierarchy, the upper agent of the root node communicates through its *upper interface* the state of the entire tree of receivers to the sender.

The second way to model RMTP captures the essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node may host multiple agents, connected to each other, each working at a different level (Figure 15). In a LAN scope, all nodes host a "local agent" component (green), similar to the "bottom agents" above, that serves as a peer in the group of all LAN nodes. The *bottom interface* used by this agent interacts with the local state. These peers exchange ACKs and arrange for message forwarding, with one of them acting as a "parent" and all other as "children". On the node hosting the "parent", a "higher-level" agent is hosted (orange); we refer to it as a "LAN agent", for there is exactly one in each LAN, and it represents the entire LAN. It connects through its bottom interface to the local agent, which is a distinguished peer in a LAN peer group, to obtain information concerning the LAN it is controlling, e.g. ACKs. These LAN agents themselves form a "higher-level" peer group. One serves as a distinguished parent node, others as subordinates. The LAN agents are communicating with each other to arrange for forwarding messages, and they jointly calculate the ACK information for the entire scope, which in

this case could be e.g. a data center in which the LANs reside. The distinguished node that hosts the parent LAN agent also hosts a yet higher-level component, call it a "data center agent". This agent could communicate with the sender, or the construction might continue further in a similar fashion. Note how in this example the peer groups defined at various levels overlap with scope boundaries.

Note also that as long as their interfaces match, each peer group could run an entirely different algorithm. We believe this power could be extremely useful in settings where local administrators control policies governing, for example, use of IP multicast and hence where different groups may need to adhere to different rules.

The issue of how to select protocols at different levels in such a way that their interfaces would match is beyond the scope of this paper. In our forthcoming paper [7], we introduce a new mechanism that could help address this issue in a more systematic manner.

To keep the presentation simple, in the model and in the examples we discussed a peer group handles recovery in a single topic. In our full design, a group of peers can handle recovery in multiple sessions at once. Throughout the lifetime of the group, peers will be instructed to begin recovery for certain sessions, at some point later they will enter the flushing phase for specific sessions (while other sessions may still be active), and may finally be requested to cease any activity for specific sessions. Accordingly, a peer, via its bottom and upper interfaces, exchanges data and requests related to multiple sessions at once. One may think of a peer as having multiple pairs of bottom and upper interfaces, each pair for a different set of sessions. Also, peers hosted at a physical node will not necessarily form a vertical, linear stack, as in our examples, the same lower-level peer may interact with two or more peers at a level above it. We omit details for clarity. All techniques that we introduced here carry over to the full design.

2.10. Implementing Recovery Algorithms

In section 2.8 we have explained how a hierarchy of recovery domains is built, such that for each session, the domains "responsible" for it form a tree. In section 2.9 we gave an example of how an algorithm such as RMTP can be modeled in our framework as a network of agents that handle the recovery tasks at various levels. A distributed recovery domain **D** in our framework will correspond to a peer group. When **D** is created at some scope **X**, the latter selects an algorithm to run in **D**, e.g. a ring or a tree, and then every sub-domain $\mathbf{D_k}$ of **D** is requested to create an agent that acts as a "peer $\mathbf{D_k}$ in group **D**". Note how the membership algorithm provides membership view at one level "above", i.e. the scope that owns a particular domain would learn about domains in all the sibling scopes. This is precisely what is required for each peer $\mathbf{D_k}$ in a group **D** to learn the membership of its group.

When the SM of a scope **X** learns that an agent should be created for one of its recovery domains $\mathbf{D_k}$ in group **D**, two things may happen. If **X** manages a single node, the agent is created locally. Otherwise, **X** delegates the task to one of its sub-scopes. As a result, the agents that serve as "peers" at the various levels are delegated to individual nodes. We thus arrive at a structure just like on Figure 18, where each node has a stack of one or more agents, each operating at a different level, linked to one another. When the node hosting a "higher-level" agent crashes, the agent is delegated to another node. Since our framework would transparently recreate channels between agents, it looks to other peers agents as if the agent lost its cached state (not permanently, for it can still query its bottom interface and talk to its peers). This requires that algorithms be defined in a way allowing peers to crash and resume with some of their state erased. Based on our experience, for a wide class of protocols this is not hard to achieve.

3. Evaluation

The need for brevity precludes a detailed discussion of the performance of our architecture. The strength of this design lies in its extensibility, ability to accommodate a wide range of transport and recovery protocols, and in facilitating the cooperation among independent parties in creating a global publish-subscribe infrastructure. Such benefits are hard to quantify. However, in certain scenarios, our approach can also greatly improve scalability. In [8], we show how we used the model and principles presented here as the basis for the design of QSM [1], a new publish-subscribe platform offering a simple ACK-based reliability and extremely scalable in multiple dimensions. We are also in the process of creating a reference implementation of the infrastructure outlined here. Ultimately, this effort will lead to a set of specifications similar to [2].

5. References

[1] K. Ostrowski, K. Birman, and A. Phanishayee, "QuickSilver Scalable Multicast". In submission, 2006.

[2] http://ifr.sap.com/ws-notification/ws-notification.pdf

[3] http://ftpna2.bea.com/pub/downloads/WS-Eventing

[4] S. Paul, and K. Sabnani, "Reliable Multicast Transport Protocol". Journal of Selected Areas in Communications (1997).

[5] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing". IEEE/ACM TONS (1996).

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". ICDCS '99.

[7] K. Ostrowski, K. Birman, "Achieving Modularity and Scalability via Typed Communication Endpoints". Forthcoming.

[8] K. Ostrowski, K. Birman, "Extensible Web Services Architecture for Notification in Large-Scale Systems (Extended Version)". Cornell University Technical Report. Forthcoming.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems

Ken Birman, Anne-Marie Kermarrec, Krzystof Ostrowski,
Marin Bertier, Danny Dolev, Robbert Van Renesse
Cornell University, Ithaca; INRIA/IRISA and IRISA/INSA, Rennes; Hebrew University, Jerusalem

Abstract

Challenges of scale have limited the development of event notification systems with strong properties, despite the urgent demand for consistency, reliability, security, and other guarantees in applications developed for sensitive tasks in large enterprises. These issues are the focus of Quicksilver, a new multicast platform targeted to large-scale deployments. An initial version of the system can support large numbers of overlapping multicast groups, high data rates and groups with large numbers of members. However, Quicksilver still requires manual help when discovering the system configuration and can't easily enforce certain types of application monitoring and integrity constraints. In this paper, we propose to extend Quicksilver by introducing gossip mechanisms, yielding a self-managed event notification platform. The two technologies are presented through a single interface and appear to end users as live distributed objects, *side-by-side with other kinds of typed components.*

1. Introduction

As we look to the next generation of distributed computing platforms, it is hard not to feel concern at the accelerating deployment of systems that will play sensitive roles, and yet will be built using fragile technologies. For example, an electronic health records system must achieve high levels of availability and consistency, be largely self-configuring, and maintain privacy and security. A typical deployment scenario involves decentralized systems linked over networks, integrating subsystems running at hospitals, other care providers, laboratories, insurance companies, pharmacies, etc. Electronic monitoring devices and other sensors running both in the hospital and at home will contribute time-sensitive data, and some therapeutic and drug delivery devices will be remotely controlled.

To reduce cost and leverage standardization, a system of this sort would probably be constructed using COTS platform technologies, such as web services. Doing so also brings productivity benefits, in the form of development tools and runtime support, and makes it easy to integrate pre-supplied functions with new application-specific ones. However, today's solutions lack the sorts of strong properties needed for sensitive uses. Our objective is to extend these platforms by adding robust tools that bridge gaps while complying with standards.

The nerve center of a modern service-oriented architecture is its event notification subsystem. Event notification services can distribute sensor readings and other kinds of updates to widely distributed system components, and can be used to replicate information where an application or a record is available at multiple locations. By decoupling publishers from subscribers, these services make it easy to upgrade an application over time and to integrate components that run on dissimilar platforms or were implemented using very different technologies. On the other hand, traditional event notification platforms lack the strong guarantees needed for medical decision making and other critical roles.

If we can create a new kind of scalable, robust event notification architecture that fits seamlessly into modern development platforms such as Windows .net or J2EE, and yet has strong properties that reduce to rigorously specified protocols that the end user can count upon and reason about, we can help application developers create robust applications for sensitive uses.

In this paper, we focus on scalability, robustness and self-management, deferring issues of security and privacy for the future. For scalable event notification with strong reliability guarantees, we've developed Quicksilver: a high-performance multicast technology that can implement a variety of reliability models, including consensus-based ones [1][2]. Traditionally, systems implementing reliable multicast have scaled poorly, but as reported below, this problem can be overcome. Moreover, although we don't tackle the question here, we believe that Quicksilver can be secured using digital certification certificates, by authenticating access to information resources, and encrypting all network traffic using per-event-channel keys that can be refreshed whenever the set of subscribers changes.

The existing version of Quicksilver is weaker with respect to self-configuration and self-management; both critical requirements for the sorts of applications we hope to support. In our target environments, the pace of reconfiguration could be very rapid: if a patient falls ill, providers might (in effect) hand the family a box full of equipment to be deployed throughout the home. Needs change as the patient's care plan evolves. Patients are moved from unit to unit. Thus one must imagine a highly dynamic, rather unpredictable environment in which the

[1] Contact: ken@cs.cornell.edu; The Cornell research group was supported by grants from AFRL/IFSE, AFOSR, NSF and the Intel Corporation.

sets of components, their configurations, and their communication patterns change constantly. Against this backdrop, we seek an event notification infrastructure that can configure itself, that can adapt as conditions evolve, and that can be leveraged to support self-configuring applications.

Fault-tolerance poses closely related problems. Today, Quicksilver offers fault-tolerance through models such as virtual synchrony, where applications are structured into groups and, if desired, will be notified when membership changes. But not all integrity constraints map easily to group membership tracking. For example, the decoupling of publisher from subscriber is advantageous from a development perspective, but sometimes correct function requires that there be an active subscriber associated with certain topics. One such case involves logging accesses to patient records for offline audits. If this functionality is implemented using event notification, it important that the logging service be running when audit events are published. Yet even if built upon a substrate such as Quicksilver, today's event notification APIs lack mechanisms to express such constraints, and hence can't trigger exceptions when they are violated.

To address self-* needs, both within Quicksilver and in applications built using it, we propose to use technology emerging from work on *gossip* protocols. Gossip encompasses a large class of protocols that exploit randomness to achieve surprising robustness under a wide range of operating conditions. They can be made self-configuring, adapt rapidly after disruption, and support a diversity of useful end-user functionality.

The integration of gossip with multicast in a single setting poses non-trivial systems-engineering challenges. Here, we propose such a unification. Although our new system is still under development, it will offer a seamless infrastructure in which Quicksilver runs side-by-side with gossip-based mechanisms to provide a self-managed scalable event notification capability. The system will expose these gossip mechanisms so that applications can exploit them directly in the same paradigm used to expose Quicksilver's multicast functionality. Here we sketch out the architecture and discuss some research challenges it poses; several appear to be of broader relevance.

The paper is structured as follows. First, we spend a moment discussing the strengths and limitations of gossip technologies. The goal is not to be exhaustive, but rather to identify styles of gossip that are both highly effective and well matched to our self-management objectives. Next, we review Cornell's new platform, Quicksilver, touching both on its scalability and its unusual embedding into the Windows .net framework. The latter topic emerges as a source of leverage in what we are now proposing to do. Finally, we explore the options for integrating the two, arriving at an architecture that (we believe) is interesting in several respects. First, it sets gossip side by side with scalable event notification. Next, the system offers an elegant embedding into Windows so that developers can benefit from that system's powerful component integration functionality and development tools (a Linux version is also under design). And finally, it suggests a path for future evolution of service-oriented architectures and standards. The paper concludes by discussing open research questions.

2. Gossip protocols

A *gossip protocol* is one with the following properties:

1. The core of the protocol involves periodic, pairwise, inter-process interactions.
2. The information exchanged during these interactions is of (small) bounded size.
3. When node *a* interacts with node *b*, the state of *a* evolves in a way that reflects the state of *b* (and vice versa). For example, if *a* pings *b* merely to measure RTT, this is not a gossip interaction.
4. Reliable communication is not assumed.
5. The frequency of the interactions is relatively low when compared to typical message latencies.
6. There is some form of randomness in peer selection.

There are three prevailing styles of gossip protocol.

1. *Dissemination (rumor-mongering) protocols*. These use gossip to spread information; they basically work by flooding nodes in the network, but in a manner that produces bounded worst-case loads:
   a. An *event dissemination* protocol runs in response to events and can be understood as using gossip to carry out multicasts, although the events don't actually trigger the gossip (since gossip runs periodically).
   b. A *background data dissemination* protocol gossips continuously to track the evolution of state at participating nodes.
2. *Anti-entropy protocols* repair replicated data by comparing replicas and reconciling differences.
3. *Aggregation protocols* compute a network-wide aggregate by sampling information at the nodes in the network and combining the values to arrive at a system-wide value – the number of nodes in the system, the sum or average of some value, etc

Our definitions are rather broad; indeed, many protocols that predate the earliest use of the term "gossip" fall within our definition. In particular, notice that a gossip substrate can "mimic" a standard routed network. That is, nodes could "gossip" about traditional point-to-point messages, in effect tunneling normal traffic through a gossip layer. Bandwidth permitting, this implies that a gossip system can potentially support any classic protocol or distributed service. Nonetheless, when we talk of gossip, we rarely intend such a broadly inclusive

interpretation. More typically we have in mind protocols that run in a regular, periodic, relatively lazy, symmetric and decentralized manner; the high degree of symmetry among nodes is particularly characteristic. To illustrate this point, consider that one could run a 2-phase commit protocol over a gossip substrate, piggybacking the messages on gossip traffic. In our view, doing so would be at odds with the spirit of the definition: there's nothing wrong with such a protocol, but it isn't gossip!

2.1 The Limitations of Gossip

The stylized manner in which we normally use gossip introduces significant limitations. First, consider the implications of the small, bounded message sizes and the relatively slow periodic message exchanges. These combine to limit the information carrying capacity of a gossip algorithm. For example, if gossip is used to disseminate information (often, in a form of flooding), the system-wide capacity for new events will be limited simply because the aggregate "bandwidth" available is bounded. The problem is that gossip protocols keep the nodes in a network busy while information spreads – typically, a process that requires O(log(n)) time. It follows that the "rate" at which events can be introduced will be proportional to 1/log(n).

The relatively slow spread of gossip can also be an obstacle. While it is common to claim that users need only tune the gossip rate to match their goals, requirement 5 complicates the picture. Gossip rates approaching the network RTT are out of the question.

Finally, gossip can be fragile in the face of malicious behavior (components that malfunction, for example by running the protocol incorrectly, disseminating incorrect data, and so forth). Recent work on BAR Gossip [21] tries to overcome some of the issues by using verifiable pseudo-random peer selection to avoid selfish and malicious behaviors. But this is just a first step.

2.2 Strengths of Gossip

Although gossip has limitations, these protocols do have substantial power. Among the most cited strengths are these:

• *Convergent consistency.* Properly designed gossip protocols, when not overwhelmed by a higher rate of incoming "events" than the information-carrying bandwidth of the underlying channels, should have a logarithmic mixing time – any new event will, with high probability, affect all nodes that need to learn about it within time logarithmic in the system size.

• *Emergent structure.* Earlier, we contrasted a classic deterministic protocol for building a spanning tree by leader-initiated flooding with a decentralized way of building such a tree using gossip. In the gossip style, the tree "emerges" from randomized pairwise interactions between peers. The term emergent structure is intended to evoke the image of a data structure that emerges with probability 1.0 in this manner. The structure may then continue to evolve over time as further gossip occurs.

• *Simplicity.* Most (but not all) gossip protocols are extremely simple and highly symmetric, with all participants running the same code.

• *Bounded load on participants.* Many classic (non-gossip) distributed protocols are criticized because they can generate high surge loads that overload individual components. Gossip is normally used in ways that produce strictly bounded worst-case loads on each component, eliminating the risk of disruptive load surges. In some situations, where network capacity is also a concern, peer-selection is further biased to control load imposed on network links.

• *Topology independence.* If running on a sufficiently connected networking substrate, and with sufficient bandwidth, a gossip protocol will often operate correctly on a great variety of underlying topologies.

• *Ease of local information discovery.* Many gossip protocols are used for purposes of discovery, for example to find a nearby resource (these are usually protocols in which gossip occurs between neighbors, not between arbitrarily distant peers). Unlike local flooding, which scales poorly, gossip would typically find local information less quickly but with bounded costs: perhaps, a constant or a delay logarithmic in the system size.

• *Robustness to transient network disruptions.* As time elapses, there are exponentially many routes by which information can flow from its source to its destinations. However, not all uses of gossip are robust in all ways. For example, unless data is self-verifying, dissemination protocols are often vulnerable to data corruption. Anti-entropy protocols may similarly be at risk if a replica becomes corrupted. And aggregation protocols are vulnerable not just to the introduction of faulty information, but also to computational errors that result in a faulty computation of the aggregate.

2.3 Appropriate roles for gossip

The foregoing discussion suggests a number of natural roles for gossip in large-scale event notification systems.

The earliest uses of gossip were to disseminate information in large-scale systems [22]. Scalability and robustness were cited as the primary benefits in these uses: the load on each node grows in a logarithmic manner as the system scales and information can be

reliability disseminated in the presence of a high proportion of node failures [20]. Such properties rely on the fact that each node samples network state randomly. This pseudo-randomness can nonetheless be controlled or "shaped". For example, Lpbcast [19] and Cyclon [15] are protocols in which each peer periodically selects another peer with which it gossips; they differ in the details of target selection, and in the way they merge information gathered through the gossip exchange with their own.

Generalizing these ideas, gossip may be used to create unstructured overlay networks, achieving properties close to those of random graphs [12]. Having used gossip to create such a graph, gossip protocols can also run over them, for example to create an overlay optimized with respect to an application-specific metric. For example, T-man builds overlays that use application-supplied quality functions to bias neighbor selection [10]. In [14], the gossip itself is biased; users with shared interests are structured into peer groups for file sharing, substantially improving response times in a search application.

Similarly, GosSkip [17] and Sub-2-Sub [13] build content-based publish-subscribe systems in which the overlay topology matches the subscription pattern. In GosSkip, subscriptions are organized into a skiplist structure so that events will be routed to interested subscribers in a logarithmic number of hops. In Sub-2-Sub, several gossip-protocols are layered to efficiently support range subscriptions. The lowest layer uses random peer sampling to ensure connectivity and robustness, a second layer creates clusters of "close" subscriptions, and the third layer structures overlapping subscriptions to ensure an exact and exhaustive dissemination of events.

This flexibility comes at a price. Gossip-based publish-subscribe overlays are often slow: the technology is wonderful for matching publishers with subscribers, but says little about getting events delivered rapidly, robustly, and with strong reliability properties. Indeed, we like to think of these kinds of applications as having two disjoint aspects: a gossip infrastructure that, in these cases, builds an overlay; and then a distinct dissemination structure that uses the overlay to reliably distribute events.

This way of thinking leads back to our current goals. We hope to systematically ask how gossip can be valuable in event-notification systems such as Quicksilver and in the applications that run over it. A number of options seem to be worth exploring. For example, as just seen, a gossip-constructed overlay network could be useful for efficient dissemination. In this case, Quicksilver itself would provide the "quality metrics" used to optimize the overlay, and the associated cost functions would reflect the mechanisms Quicksilver uses for dissemination and for recovery of lost packets.

More broadly, we hope to use gossip to materialize a form of distributed "picture" of the application network, which would become an input to an auto-configuration application that would generate configuration files. These would advise the end-user application (in addition to the Quicksilver event notification infrastructure) of the topology on which it should operate and the appropriate parameter settings to use. Later, as conditions evolve, the same approach could be used to reconfigure the running system so as to repair damage caused by a failure, or to integrate new components with the existing infrastructure.

Another possible role for gossip would be to track overall loads, loss rates and other status in the system. We have experience with a gossip-based system used for this purpose. Astrolabe is a distributed monitoring and data mining system that uses gossip to construct a virtual hierarchical database that can be queried much like a normal database [5]. The database is extremely useful for self-optimization and problem diagnosis. Because Astrolabe is fully replicated it has no single point of failure or load-related hot-spots, and the underlying gossip protocol remains robust even under stress that can shut down most other system functionality. In our new system, we believe aggregation mechanisms can play even more roles, including parameter setting and dynamic adaptation [11]. Aggregation can even be used for resource allocation, for example by using gossip to sort peers according to an application-specific metric [16].

Finally, we will use gossip to support background diffusion of system information that won't be needed immediately, but could be of high value "later". A tool permitting discovery of available information sources would be one possible use for such a mechanism. Other possibilities include mechanisms for tracking contact nodes or other services, finding information stored elsewhere in the network, etc. By using gossip to disseminate the underlying information, we can be certain that data will get through even if the system configuration changes (or is disrupted), and hence will be available when and where needed.

To exploit these kinds of gossip mechanisms, we need to tackle some significant software engineering issues that prior work has largely overlooked. To make gossip useful as a tool, one needs appropriate embeddings of these abstractions into the runtime environment. For these purposes, we propose to extend a feature of Cornell's Quicksilver platform, discussed below.

3. Quicksilver

Cornell's Quicksilver project [3][4] offers a scalable event notification infrastructure that can support strong properties on a per-topic basis. An application can subscribe to large numbers of communication channels, with the properties of each channel matched to the data it carries. Krzysztof Ostrowski is the lead architect and developer for Quicksilver, in collaboration with Ken Birman, Danny Dolev and Robbert van Renesse. We start

by reviewing prior work on Quicksilver, and then suggest some of the extensions our new effort will explore.

A key objective for Quicksilver is scalability in multiple dimensions: numbers of applications using the platform, numbers of event channels to which each application subscribes, data rates, tolerance of disruption, etc. Our underlying premise is that inadequate scalability has limited the uptake of group-multicast in general, and has prevented its widespread use in support of event notification. This sometimes manifests itself through throughput that degrades gracefully as the system is deployed into a larger setting, but more dramatic consequences are also observed. For example, many large-scale event notification platforms become unstable in large deployments, oscillating from very low throughput to overwhelmingly high data rates in which traffic generated by the platform can actually shut down the communications bus by swamping it with data, retransmissions, nack and ack messages and other forms of overhead – a so called broadcast storm effect. In designing Quicksilver, our goal was to demonstrate stability in this problematic domain.

This is not the right setting for a detailed discussion of the Quicksilver architecture. Instead, we summarize some key ideas very briefly:

- *Separation of concerns.* Quicksilver treats event dissemination separately from recovery of lost packets, flow control, and imp lementation of stronger consistency ("properties").
- *Regions of overlap.* A single node will often subscribe to many event channels. If each channel is treated as a separate multicast group, one encounters obvious problems of scale. Accordingly, Quicksilver maps from overlapping channels down to *regions*, defined to be sets of nodes with similar subscriptions. Dissemination is on a per-region basis; recovery is done in an aggregated manner over regions, etc.
- *Scalable recovery.* Quicksilver uses a novel hierarchy of token rings to achieve scalable detection of lost packets and, when possible, to recover data between peers in a region, offloading work from the sender.
- *Per-channel reliability properties.* The reliability properties of each channel can be matched to its role.
- *Managed runtime environment.* Quicksilver runs in managed settings, allowing it to leverage strong type checking, memory management, etc.

Details of the architecture and protocols appear in [3][4].

Quicksilver has been running since June 2006. For the moment, all our users are building datacenters – WAN scenarios are a goal once the new gossip-based mechanisms are available, but the current system doesn't run in WAN settings. In our datacenter experiments, we've set up groups with up to 200 nodes (larger runs are planned), than subjected them to extremely high throughputs and injected various forms of stress.

Up to the present, we have seen only minimal throughput degradation and no signs of instability or throughput fluctuations even in the largest configurations. In contrast, such problems are easy to provoke in most existing technologies for multicast in the same settings, even with much smaller groups of just 50 to 75 members [2]. Quicksilver can saturate a 100Mbit ethernet interconnect with just 20-40% CPU loads on the inexpensive PC's making up our test cluster; experiments with our prior systems peaked at about a tenth these data rates and generated much heavier loads. Perhaps most important, processes are able to access large numbers of groups. For this reason, when used to support event notification, Quicksilver can maintain steady performance even when each process joins as many as 8000 separate event channels [3][4]. Obviously, this capsule summary oversimplifies in some important ways (in particular, not all configurations of processes and event streams are supported), but they do give a sense of what the system should be able to achieve.

Of primary relevance here is the manner in which Quicksilver embeds event notification channels into Windows. Traditionally, event notification platforms have been treated as a free-standing technology that lives separately from the operating system. Quicksilver can be used this way too, through a conventional publish-subscribe infrastructure that generalizes the web services eventing standards (in [6] we discuss our reasons for extending these standards rather than working entirely within ws-notification or ws-eventing).

But Quicksilver also offers a second, deeper embedding into Windows in which event notification channels can be accessed either as a new kind of distributed *live object* visible in the file system side-by-side with other named objects. These objects are best understood as distributed abstract data types. A program accesses such an object much as it would access a file in Windows: given appropriate permissions, it can open the object, read the current state, and will receive events as the state is subsequently updated. This, however, is an illusion: the "object" is really an event channel, and the state is a checkpoint produced by some existing subscriber when a new program subscribes. State persistence is available, but optional.

We've emphasized the similarity between the way that a system such as Windows understands file "types" as an association between the data in some object and the programs that implement operations on that kind of object, and the way that Quicksilver associates a type with each event notification channel. For Quicksilver, the type corresponds to an object class, but also is associated with a definition of the properties the channel should implement. The effect is to confer a distributed semantics on the group of objects as a whole. The approach is flexible enough to support weak properties such as best-effort notification, stronger consensus-based properties

such as the virtual synchrony model, or even very strong models such as transactional 1-copy serializability. Quicksilver implements a domain-specific programming language within which the properties associated with each event channel can be specified. The system basically compiles these property definitions into pseudo-code which it can execute to achieve the desired behavior.

4. A unified platform

For our purposes, the key point of leverage involves the embedding of Quicksilver's live objects (event channels) into Windows. Consider the integration of abstract data types such as Excel spreadsheets or Word documents into the Windows file system. Windows uses the filename extension to understand the "type" of the object, allowing it to interpret operations on the object as method invocations on an appropriate application program. Web services standards are used in conjunction with these componentization mechanisms: active components such as the Excel application register their interfaces using the Web Services framework built into .net, at which point the Windows platform can function as a component integration environment using Web services standards and protocols to perform tasks such as method invocation. Of course, this component-to-component type system is somewhat primitive, but one could imagine taking the idea much further; indeed, there are projects underway at Microsoft to do just that. It isn't unreasonable to imagine that future versions of Windows will incorporate a full-fledged distributed type system at the component level.

As suggested above, Quicksilver extends Windows to support abstract data types with "live" content, and allows a variety of event stream providers to support the live aspects of the abstraction. A Quicksilver event notification channel has a name that can be visible in the file system name space, and a type, corresponding to the properties associated with the event channel. When an application binds itself to an event channel, Windows passes the binding event to Quicksilver, and we can perform type compatibility checking, or can even perform some kinds of dynamic type coercion (for example by introducing an encryption/decryption layer in order to integrate a component that doesn't support encryption with an event channel that requires stronger forms of security). The same mechanisms also work from the Windows shell: if a user right-clicks on a Quicksilver event channel, the shell extensions framework passes us the request. Quicksilver can then identify applications that can connect to this kind of channel, and can even generate dynamically created virtual folders, for example displaying thumbnail-size images from a video streaming application.

Quicksilver is thus on a path towards the same kind of tight integration with Quicksilver event streams as is seen with other Windows communications options such as DCOM. The approach enables developers to leverage existing Windows application development and debugging tools while benefiting from co-existence in a managed framework. If Windows evolves in the manner currently anticipated, type checking will become possible even across component boundaries. Because Quicksilver uses the CLR memory management layer, no copying occurs when a large object is multicast. Of course, such a positioning of the technology also brings challenges of its own (for example, to maximize performance in a managed environment requires protocol designs quite different from those one uses in a Linux/C multicast implementation [3]) but the problems are solvable and we believe the result is well worth the effort. We should comment that although Windows is our initial target, everything we are doing should port (using Mono) to Linux and would then be accessible from J2EE or even Corba applications.

This, then, is the core contribution of the present paper: a vision of how one might unify these three worlds: objects in a platform such as Windows on the one hand, and both gossip and of scalable event notification on the other, all in a single framework. A first step towards this vision requires that the Quicksilver multicast framework be separated from the mechanisms that embed Quicksilver objects into Windows; Ostrowski is already developing this capability as part of version 2.0 of the system. As is the case in Quicksilver today, the basic abstraction will be that of a distributed object having a "state" and an associated event stream. However, rather than assuming that the live content is transported by Quicksilver's reliable multicast protocols, there will be at least two possible communication infrastructures – the other being gossip-based. Down the road one might imagine additional options, such as an IP-TV streaming layer, or one focused on real-time communication.

Thus, referring back to the examples of gossip-based mechanisms mentioned in Section 3, one could build a gossip-based topology and configuration discovery service that, in effect, produces an annotated picture of the state of the system. An end-user could access that picture by clicking on an associated file name; doing so would launch some sort of browser capable of visualizing this kind of information and might let the user explore the network, for example to pin down a bottleneck that is impacting performance. Application programs could use the picture to configure themselves. And Quicksilver's event notification infrastructure could use that picture to construct overlays for disseminating events that use IP multicast when possible, but tunnel data through overlay trees where IP multicast is not feasible (and these same overlay networks would also be available to application designers, through some form of abstract data type). The remarkable robustness of the gossip protocols ensures that even when all else is disrupted, applications can still

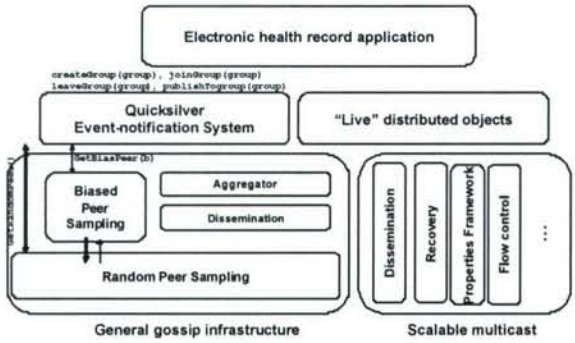


Figure 1: Overall System Architecture

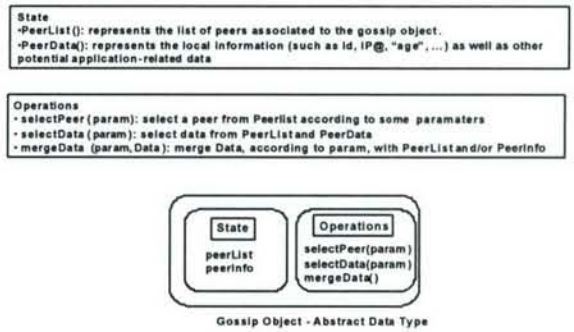


Figure 2: Generalized Implementation of a Gossip Object

monitor the system to set parameters, configure themselves, and adapt when conditions change.

But we believe we can do more than to simply import gossip functionality into Quicksilver. Gossip systems of the types we reviewed share substantial commonalities across their various presentations. For example, many gossip mechanisms require random peer selection, either within the full system (a kind of *anycast*) or within a set of neighbors of a node (a local variant on *anycast*). The thinking is that this and other low-level primitives can be standardized within the gossip subsystem, and then reused across gossip-based objects. Doing so poses interesting research challenges: if a single object employs *anycast*, one can implement a "greedy" solution. But suppose that on some single node there are tens or even hundreds of gossip-based objects, all using *anycast*. Could we aggregate, so that a single message can carry information on behalf of multiple objects?

One can pose similar questions at a higher level. Many gossip algorithms are highly stylized: the nature of a gossip exchange is rather similar across most gossip-based mechanisms, even if the details of what "state" is exchanged and how it is "merged" differ. This immediately suggests that one might design an abstract gossip state-machine that could be instantiated in multiple objects, parameterized with appropriate state marshalling and merge functions.

The resulting architecture is summarized in Figures 1 and 2. Figure 1 illustrates the overall system architecture, with the gossip infrastructure hosted side-by-side with the scalable multicast infrastructure and accessed either through a generalized publish-subscribe interface, or in the form of live distributed objects. As noted earlier, internal details for Quicksilver can be found in [3] and will not be repeated here. Figure 2 gives some additional detail for the gossip infrastructure.

5. Electronic health record example

We conclude the discussion by revisiting our electronic health record example, assuming now that the gossip mechanisms and the Quicksilver-based event notification solution are available side-by-side.

Let's start with roles for the gossip mechanisms. For the time being, we've decided to focus on uses in which the gossip components will be simple enough so that we can verify correctness, able to "sanity check" data collected from the environment, and unlikely to come under attack; these assumptions mitigate the security concerns mentioned earlier. For example, with gossip it isn't difficult to build a system that can track locations of system components: servers, client platforms, sensors, other devices. When a change occurs, the updated configuration should become visible with delay proportional to the log of the size of the system – in the scenarios we have in mind case, probably within 10 or 15 rounds of gossip. This capability could be the basis for a highly robust plug-and-play technology, whereby the health-care system would adapt in tens of seconds as conditions evolve. Although such a system might collect incorrect information about a platform that has some form of scrambled configuration state, the "damage" would be limited to the annotation of that component on the map, and the gossip objects can be designed to sense and reject implausible inputs.

Gossip could also be used to monitor system invariants (such as: "there should always be at least one instance of the auditing service"). Here, Quicksilver's notion of membership offers very rapid event detection and reaction, but if enough damage occurs while the system is running to seriously disrupt event notification, the gossip layer could guide a timely discovery of the problem and dynamic repair or adjustment of the parameters. The remarkable robustness of gossip mechanisms gives us reason for confidence that they will be able to continue to operate reliably even when other infrastructure components are severely degraded by a disruptive event.

Gossip can also be used to help system components connect themselves in appropriate ways. For example, a component might keep track of the locations of the various servers so that in the event of a fault that prevents

connection to one server, the clients using it can seamlessly roll over to others offering backup functionality. When the first server recovers, the clients can shift back. Gossip mechanisms can be used to monitor system health, assisting managers in diagnosing and repairing problems that arise because of software bugs or other disruptive events. If a firewall or server comes under attack (or just becomes overloaded), gossip based tracking mechanisms can help client systems discover the problem, identify fall-back options, and gracefully adapt.

Gossip also offers an antidote to certain kinds of fragility. For example, suppose that we want to track the physical location of patients in our hospital complex. In the most obvious standard implementation of an electronic health record system, one would probably place some sort of active component on the patient's gown or bed; it would continuously track its own location (somehow) and report that data to the central database. With gossip, new and potentially more robust options arise. Now, client systems can gossip with one-another about patient "sightings". With many observers and many paths by which information can spread, we obtain a patient location-tracking database at low cost, and with guarantees of extremely robust behavior even in the event of a disruptive condition, such as a malfunctioning application that generates extremely high network loads and loss rates. (Recall from our discussion of Astrolabe that a gossip management infrastructure might help in this case too, by assisting the system administrator in localizing the problem).

What about high-speed event notification and streaming? Our system could exploit this functionality in a great many ways. If we assume that health care records are, in effect, replicated throughout the system as a whole, when an update occurs, it will be important to consistently update all copies. Here we see a form of event notification that requires relatively strong reliability and delivery semantics – corresponding to a consensus-based model such as virtual synchrony or state machine replication, both available within Quicksilver as group "types". Event notification can support a publish-subscribe relationship between the database servers in the hospital and client systems operated in private practices and other satellite locations. Bedside or nursing station display systems may need to be refreshed. Similarly, if the update is relevant to a patient's prescriptions, the event might be pushed out to participating pharmacies. One can also imagine high-throughput event channels. For example, television cameras and other sensors monitoring infants in a neo-natal unit could stream images to the nursing station; pediatricians would be able to subscribe as necessary to keep an eye on their patients: a robust, scalable IP-TV architecture

The Quicksilver properties mechanisms would be beneficial here, by permitting the system to match the properties of each type of event channel, or live object, to the requirements associated with that category of object. In fact we doubt that there would be a huge number of cases, but there are clearly subsystems that would value real-time data delivery over other guarantees, subsystems that need the sorts of consistency afforded by virtual synchrony or state machine replication, and subsystems that need transactional "ACID" properties. These can all be supported, side-by-side, on a per-event-channel basis.

These example illustrates a point worth reiterating:b y using the publish-subscribe paradigm, the publishing side of the enterprise can be designed independently from the data consuming side; both can be incrementally extended over time as new applications are added, and will automatically accommodate varying runtime configurations. In effect, we are able to separate the information representation standards used within the system (including the hierarchy of topics) from the data sources and the data consumers. The communications infrastructure provides the needed guarantees, and when a new component is introduced, existing event-generating applications don't need to be modified. Because Quicksilver has a strong notion of types associated with event channels and live objects, we can do far more type checking than is traditionally feasible in publish-subscribe settings. For example, we can potentially ensure that the properties of a channel match the expectations of the application that binds itself to that channel. Moreover, to the extent that we need instant detection and reaction to a failure, because Quicksilver extends the publish-subscribe eventing model to also offer (optional) information about subscription changes when processes join and leave a channel, all sorts of rapid fault-tolerance mechanisms can be implemented.

We've avoided discussion of privacy and security issues, despite their central importance in electronic health care systems. This is in part because Quicksilver currently lacks a comprehensive security architecture, although we do have some ideas for how we might build one. Our thinking is to focus on capabilities enabled by the secure replication of security keys using the algorithms of Reiter [8][9] or Rodeh [7]; these offer ways to refresh keys when the set of nodes in the replication group (the event channel) changes because of a failure or a join. However, prior research has never explored scalability implications of these kinds of secure key replication schemes, and we believe the topic will require a substantial research effort to fully resolve. Use of security keys in gossip settings represents an additional intriguing option for study.

7. Research topics

Our vision raises a number of questions:

1. Given a proposed large-scale application, what is the most effective development methodology for mapping it down to application-specific functionality, as opposed to platform-supplied functionality? How should the developer make decisions concerning the aspects that are best matched to gossip communication, those best matched to event notification, and those that require hand-coded logic? Given that both gossip and event notification systems can support "guaranteed" properties, how should the developer decide which properties are needed by a given application, and how best to achieve them? Is there are large-scale methodology for specification of overall properties of a complex system that might lend itself to a formal verification process analogous to the ones used to reason about and ultimately prove correctness for non-distributed systems? Can the properties mechanisms used in Quicksilver today be extended to include gossip protocols?
2. If a single computer system supports multiple "live" data objects, high performance often requires that protocols be designed to amortize costs. Much of the innovation in Quicksilver is at this level: the system looks for ways to disseminate data, recover from packet loss and control data rates that are aggregated across potentially huge numbers of objects. When we introduce new classes of objects supported by gossip, the gossip infrastructure will need to address similar questions.
3. We alluded to the need to secure the platform, and to the risk that gossip mechanisms might be incapacitated by certain kinds of malicious behaviors. Our architecture poses significant opportunities for research on security, ranging from questions of precisely how one might secure a gossip protocol to broader issues of scalability that arise if an application subscribes to a large number of secured objects. How should one secure a high-speed event channel? What issues arise as one scales a security abstraction in a setting where each separate event channel or live object might have its own security requirements?
4. The creation of appropriate abstractions for the gossip infrastructure is an important challenge. At the lowest level, one imagines mechanisms for random peer selection, state exchange and merge, aggregation, etc. Ideally, these should be highly standardized. Yet some gossip protocols bias peer selection, implement "tricky" state exchange/merge mechanisms, or perform aggregation in unusual ways. Needed is a platform that can function well as a black box, and yet that can also expose functionality as needed.
5. We need to better understand the correct set of gossip mechanisms needed for purposes of self-management and self-configuration in Quicksilver. The modern internet is complex, and while it is easy to evoke a vision of an autonomic infrastructure that can support plug-and-play behavior in almost arbitrary settings, implementing that vision is quite a different matter.
6. Applications running on the event notification infrastructure will also need self-management and self-configuration functionality. Quicksilver's needs are somewhat peculiar to its role; will the same autonomic mechanisms that work for Quicksilver be adequate for other purposes, or are other kinds of gossip tools needed?
7. Obtaining high performance in large-scale settings that involve managed frameworks (C# in .net, in our case) is surprisingly hard [3]. It is likely that we will need to overcome similar challenges as we implement a gossip-based infrastructure and then tune it to cooperate cleanly with Quicksilver.
8. We commented that one key to scalability in Quicksilver is the mapping of event channels down to regions of approximate overlap – sets of nodes with similar subscription sets. A basic assumption underlying the system is that this can actually be done and that large systems will exhibit high degrees of overlap, or at least that they can be designed to have this property. But how can overlap regions be discovered in the first place? We are thinking that gossip mechanisms could be very useful in discovering applications and their "potential" subscription sets, enabling an offline analysis (perhaps with a human designer in the loop) to identify regions of overlap and configure Quicksilver. In contrast, the alternative of trying to discover regions at runtime by analysis of subscription patterns as programs come and go raises a number of thorny problems and may not be the best approach.

9. Conclusions

Scalable event notification systems capable of offering strong properties may be the key to enabling a new generation of trustworthy distributed applications, but only if they can be integrated naturally into the most powerful development environments and made autonomic: self-monitoring, self-configuring, and self-managing. For these latter purposes, we propose to build a new kind of distributed abstraction that embeds into Windows much like a typed object, but can be supported either by Quicksilver's scalable event architecture or by gossip-based protocols. A system realizing this vision is now under joint development at IRISA/INRIA in Rennes and at Cornell University.

7. References

[1] Reliable Distributed Systems Technologies, Web Services, and Applications. Birman, Kenneth P.

2005, XXXVI, 668 p. 145 illus., Hardcover ISBN: 0-387-21509-3

[2] A Review of Experiences with Reliable Multicast. K. P. Birman Software Practice and Experience Vol. 29, No. 9, pp, 741-774, July 1999

[3] Implementing Scalable Publish-Subscribe in a Managed Environment. Krzysztof Ostrowski, Ken Birman. In Submission (November, 2006).

[4] QuickSilver Scalable Multicast. Krzysztof Ostrowski, Ken Birman, and Amar Phanishayee. Cornell University Technical Report TR2006-2063 (April, 2006).

[5] Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. Robbert van Renesse, Kenneth Birman and Werner Vogels. ACM Transactions on Computer Systems, May 2003, Vol.21, No. 2, pp 164-206

[6] Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. Submitted to International Journal of Web Services Research.

[7] The Architecture and Performance of the Security Protocols in the Ensemble Group Communication System. Ohad Rodeh, Ken Birman, Danny Dolev. Journal of ACM Transactions on Information Systems and Security (TISSEC). Vol. 4, No 3, pp 289-319, Aug 2001

[8] A Security Architecture for Fault-Tolerant Systems. Michael K. Reiter, Kenneth P. Birman, Robbert van Renesse. ACM Trans. Comput. Syst. 12(4): 340-371 (1994)

[9] How to Securely Replicate Services. Michael K. Reiter, Kenneth P. Birman. ACM Trans. Program. Lang. Syst. 16(3): 986-1009 (1994)

[10] T-Man: Gossip-based overlay topology management. Mark Jelasity and Ozalp Babaoglu. In ESOA 2005, Revised Selected Papers, vol 3910 of LNCS, 1-15.

[11] Gossip-based aggregation in large dynamic networks. Mark Jelasity, Alberto Montresor and Ozalp Babaoglu. ACM Transactions on Computer Systems, 23(3): 219-252, August 2005.

[12] The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. Mark Jelasity, Rashid Guerraoui, Anne-Marie Kermarrec, Marteen van Steen. Middleware 2004, volume 3231 of LNCS, 79-98, Springer-Verlag, 2004.

[13] Sub-2-Sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. Spryros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec and Maarten van Steen. Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS), Santa-Barbara, CA, February 2006.

[14] Epidemic-style Management of Semantic Overlays for Content-based Searching. Spyros Voulgaris and Maarten van Steen, Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), Lisbon, Portugal, August 2005.

[15] CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Spyros Voulgaris, Daniela Gavidia, Maarten van Steen. Journal of Network and Systems Management, vol. 13(2):197-217.

[16] Ordered Slicing of Very Large-Scale Overlay Networks. Mark Jelasity and Anne-Marie Kermarrec. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, 2006.

[17] GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles. Rachid Guerraoui, Sidath Handurukande, Kevin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant and Etienne Riviere. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, September, 2006.

[18] From Epidemics to Distributed Computing. Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. IEEE Computer, 37(5):60-67, May 2004.

[19] Lightweight Probabilistic Broadcast. Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. ACM Transaction on Computer Systems, 21(4), November 2003.

[20] Probabilistic Reliable Dissemination in Large-Scale Systems. Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. IEEE Transactions on Parallel and Distributed Systems, 14(3), March 2003.

[21] BAR Gossip. Harry Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, Mike Dahlin, Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI), Nov 2006 .

[22] Epidemic algorithms for replicated database maintenance. Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry. ACM SIGOPS Operating Systems Review Volume 22 , Issue 1 (Jan., 1988), 8 - 32

Ricochet: Lateral Error Correction for Time-Critical Multicast

Mahesh Balakrishnan[†], Ken Birman[†], Amar Phanishayee[‡], Stefan Pleisch[†]
[†]*Cornell University and* [‡]*Carnegie Mellon University*
{*mahesh,ken,pleisch*}*@cs.cornell.edu, amarp+@cs.cmu.edu*

Abstract

Ricochet is a low-latency reliable multicast protocol designed for time-critical clustered applications. It uses IP Multicast to transmit data and recovers from packet loss in end-hosts using Lateral Error Correction (LEC), a novel repair mechanism in which XORs are exchanged between receivers and combined across overlapping groups. In datacenters and clusters, application needs frequently dictate large numbers of fine-grained overlapping multicast groups. Existing multicast reliability schemes scale poorly in such settings, providing latency of packet recovery that depends inversely on the data rate within a single group: the lower the data rate, the longer it takes to recover lost packets. LEC is insensitive to the rate of data in any one group and allows each node to split its bandwidth between hundreds to thousands of fine-grained multicast groups without sacrificing timely packet recovery. As a result, Ricochet provides developers with a scalable, reliable and fast multicast primitive to layer under high-level abstractions such as publish-subscribe, group communication and replicated service/object infrastructures. We evaluate Ricochet on a 64-node cluster with up to 1024 groups per node: under various loss rates, it recovers almost all packets using LEC in tens of milliseconds and the remainder with reactive traffic within 200 milliseconds.

1 Introduction

Clusters and datacenters play an increasingly important role in the contemporary computing spectrum, providing back-end computing and storage for a wide range of applications. The modern datacenter is typically composed of hundreds to thousands of inexpensive commodity blade-servers, networked via fast, dedicated interconnects. The software stack running on a single blade-server is a brew of off-the-shelf software: commercial operating systems, proprietary middleware, managed run-time environments and virtual machines, all standardized to reduce complexity and mitigate maintenance costs.

The last decade has seen the migration of time-critical applications to commodity clusters. Application domains ranging from computational finance to air-traffic control and military communication have been driven by scalability and cost concerns to abandon traditional real-time environments for COTS datacenters. In the process, they give up conservative - and arguably unnecessary - guarantees of real-time performance for the promise of massive scalability and multiple nines of timely availability, all at a fraction of the running cost. Delivering on this promise within expanding and increasingly complex datacenters is a non-trivial task, and a wealth of commercial technology has emerged to support clustered applications.

At the heart of commercial datacenter software is *reliable multicast* — used by publish-subscribe and data distribution layers [5, 7] to spread data through clusters at high speeds, by clustered application servers [1, 4, 3] to communicate state, updates and heartbeats between server instances, and by distributed caching infrastructures [2, 6] to rapidly update cached data. The multicast technology used in contemporary industrial products is derivative of protocols developed by academic researchers over the last two decades, aimed at scaling metrics like throughput or latency across dimensions as varied as group size [10, 17], numbers of senders [9], node and network heterogeneity [12], or geographical and routing distance [18, 21]. However, these protocols were primarily designed to extend the reach of multicast to massive networks; they are not optimized for the failure modes of datacenters and may be unstable, inefficient and ineffective when retrofitted to clustered settings. Crucially, they are not designed to cope with the unique scalability demands of time-critical fault-tolerant applications.

We posit that a vital dimension of scalability for clustered applications is the *number of groups* in the system. All the uses of multicast mentioned above induce large numbers of overlapping groups. For example, a computational finance calculator that uses a topic-based pub-sub system to subscribe to a fraction of the equities on the stock market will end up belonging in many multicast groups. Multiple such applications within a datacenter - each subscribing to different sets of equities - can result in arbitrary patterns of group overlap. Similarly, data caching or replication at fine granularity can result in a single node hosting many data items. Replication driven by high-level objectives such as locality, load-balancing or fault-tolerance can lead to distinct overlapping replica sets - and hence, multicast groups - for each item.

In this paper, we propose Ricochet, a time-critical re-

liable multicast protocol designed to perform well in the multicast patterns induced by clustered applications. Ricochet uses IP Multicast [15] to transmit data and recovers lost packets using *Lateral Error Correction* (LEC), a novel error correction mechanism in which XOR repair packets are probabilistically exchanged between receivers and combined across overlapping multicast groups. The latency of loss recovery in LEC depends inversely on the aggregate rate of data in the system, rather than the rate in any one group. It performs equally well in any arbitrary configuration and cardinality of group overlap, allowing Ricochet to scale to massive numbers of groups while retaining the best characteristics of state-of-the-art multicast technology: even distribution of responsibility among receivers, insensitivity to group size, stable proactive overhead and graceful degradation of performance in the face of increasing loss rates.

1.1 Contributions

- We argue that a critical dimension of scalability for multicast in clustered settings is the number of groups in the system.
- We show that existing reliable multicast protocols have recovery latency characteristics that are inversely dependent on the data rate in a group, and do not perform well when each node is in many low-rate multicast groups.
- We propose Lateral Error Correction, a new reliability mechanism that allows packet recovery latency to be independent of per-group data rate by intelligently combining the repair traffic of multiple groups. We describe the design and implementation of Ricochet, a reliable multicast protocol that uses LEC to achieve massive scalability in the number of groups in the system.
- We extensively evaluate the Ricochet implementation on a 64-node cluster, showing that it performs well with different loss rates, tolerates bursty loss patterns, and is relatively insensitive to grouping patterns and overlaps - providing recovery characteristics that degrade gracefully with the number of groups in the system, as well as other conventional dimensions of scalability.

2 System Model

We consider patterns of multicast usage where each node is in many different groups of small to medium size (10 to 50 nodes). Following the IP Multicast model, a group is defined as a set of receivers for multicast data, and senders do not have to belong to the group to send to it. We expect each node to receive data from a large set of distinct senders, across all the groups it belongs to.

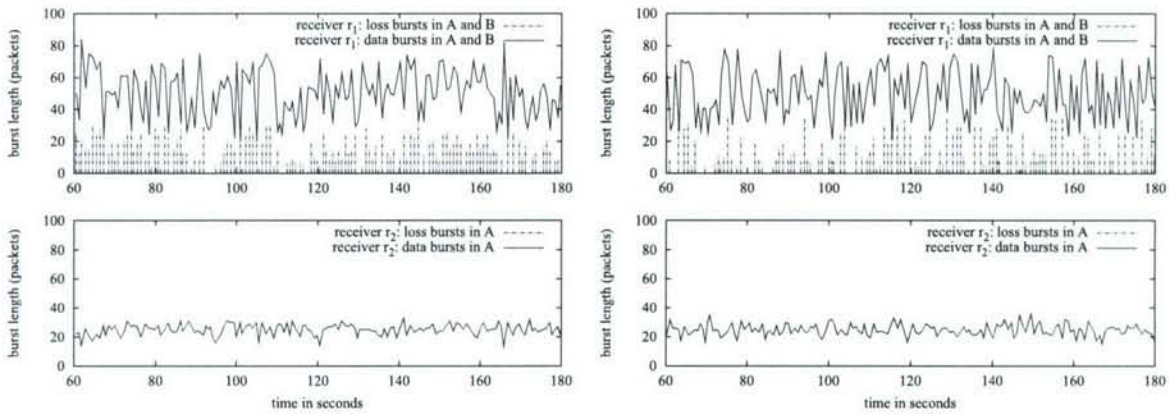
Where does Loss occur in a Datacenter? Datacenter networks have flat routing structures with no more than two or three hops on any end-to-end path. They are typically over-provisioned and of high quality, and packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are inexpensive and easily overloaded; even with high-capacity network interfaces, the commodity OS often drops packets due to buffer overflows caused by traffic spikes or high-priority threads occupying the CPU. Hence, our loss model is one of short packet bursts dropped at the end-host receivers at varying loss rates.

Figure 1 strongly indicates that loss in a datacenter is (a) bursty and (b) independent across end-hosts. In this experiment, a receiver r_1 joins two multicast groups A and B, and another receiver r_2 in the same switching segment joins only group A. From a sender located multiple switches away on the network, we send per-second data bursts of around 25 1KB packets to group A and simultaneously send a burst of 0-50 packets to group B, and measure packet loss at both receivers. We ran this experiment on two networks: a 64-node cluster at Cornell with 1.3 Ghz receivers and the Emulab testbed at Utah with 2 Ghz receivers, all nodes running Linux 2.6.12.

The top graphs in Figure 1 show the traffic bursts and loss bursts at receiver r_1, and the bottom graphs show the same information for r_2. We can see that r_1 gets overloaded and drops packets in bursts of size 1-30 packets, whereas r_2 does not drop any packets — importantly, around 30% of the packets dropped by r_1 are in group A, which is common to both receivers. Hence, loss is both bursty and independent across nodes. Together, these graphs indicate strongly that loss occurs due to buffer overflows at receiver r_1.

The example in Figure 1 is simplistic - each incoming burst of traffic arrives at the receiver within a small number of milliseconds - but conveys a powerful message: it is very easy to trigger significant bursty loss at datacenter end-hosts. The receivers in these experiments were running empty and draining packets continuously out of the kernel, with zero contention for the CPU or the network, whereas the settings of interest to us involve time-critical, possibly CPU-intensive applications running on top of the communication stack.

Further, we expect multi-group settings to intrinsically exhibit bursty incoming traffic of the kind emulated in this experiment — each node in the system receives data from multiple senders in multiple groups and it is likely that the inter-arrival time of data packets at a node will vary widely, even if the traffic rate at one sender or group is steady. In some cases, burstiness of traffic could also occur due to time-critical application behavior - for example, imagine an update in the value of a stock quote triggering off activity in several system components, which then multicast information to a replicated central data-



(a) Cornell 64-node Cluster

(b) Utah Emulab Testbed

Figure 1: Datacenter Loss is bursty and uncorrelated across nodes: receiver r_1 (top) joins groups A and B and exhibits bursty loss, whereas receiver r_2 (bottom) joins only group A and experiences zero loss.

store. If we assume that each time-critical component processes the update within a few hundred microseconds, and that inter-node socket-to-socket latency is around fifty microseconds (an actual number from our experimental cluster), the central datastore could easily see a sub-millisecond burst of traffic. In this case, the componentized structure of the application resulted in bursty traffic; in other scenarios, the application domain could be intrinsically prone to bursty input. For example, a financial calculator tracking a set of hundred equities with correlated movements might expect to receive a burst of a hundred packets in multiple groups almost instantaneously.

3 The Design of a Time-Critical Multicast Primitive

In recent years, multicast research has focused almost exclusively on application-level routing mechanisms, or overlay networks ([13] is one example), designed to operate in the wide-area without any existing router support. The need for overlay multicast stems from the lack of IP Multicast coverage in the modern internet, which in turn reflects concerns of administration complexity, scalability, and the risk of multicast 'storms' caused by misbehaving nodes. However, the homogeneity and comparatively limited size of datacenter networks pose few scalability and administration challenges to IP Multicast, making it a viable and attractive option in such settings. In this paper, we restrict ourselves to a more traditional definition of 'reliable multicast', as a reliability layer over IP Multicast. Given that the selection of datacenter hardware is typically influenced by commercial constraints, we believe that any viable solution for this context must be able to run on any mix of existing commodity routers and OS software; hence, we focus exclusively on mechanisms that act at the application-level, ruling out schemes which require router modification, such as PGM [19].

3.1 The Timeliness of (Scalable) Reliable Multicast Protocols

Reliable multicast protocols typically consist of three logical phases: *transmission* of the packet, *discovery* of packet loss, and *recovery* from it. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols, the time taken to discover packet loss dominates recovery latency heavily in the kind of settings we are interested in. The key insight is that *the discovery latency of reliable multicast protocols is usually inversely dependent on data rate*: for existing protocols, the rate of outgoing data at a single sender in a single group. Existing schemes for reliability in multicast can be roughly divided into the following categories:

ACK/timeout: RMTP [21], RMTP-II [22]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [21]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger time-outs at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

Gossip-Based: Bimodal Multicast [10], lpbcast [17]. Receivers periodically gossip histories of received packets

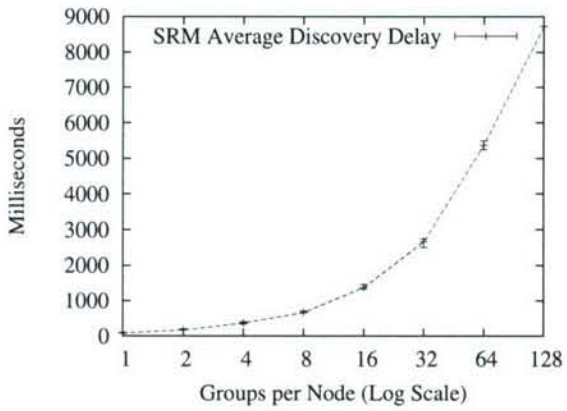


Figure 2: SRM's Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs.

with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery latency is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [10]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact.

NAK/Sender-based Sequencing: SRM [18]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery latency is thus proportional to the inter-send time at any single sender to a single group - a receiver can't discover a loss in a group until it receives the next packet from the same sender to that group - and consequently depends on the sender's data transmission rate to the group. To illustrate this point, we measured the performance of SRM as we increased the number of groups each node belonged in, keeping the throughput in the system constant by reducing the data rate within each group - as Figure 2 shows, discovery latency of lost packets degrades linearly as each node's bandwidth is increasingly fragmented and each group's rate goes down, increasing the time between two consecutive sends by a sender to the same group. Once discovery occurs in SRM, lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

Sender-based FEC [20, 23]: Forward Error Correction schemes involve multicasting redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating c repair packets for every r data packets, such that any r of the combined set of $r + c$ data and repair packets is sufficient to recover the original r data packets; we term this (r, c) parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out r data packets to that group. Fast, efficient encodings such as Tornado codes [11] make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, software distribution or internet radio.

Receiver-based FEC [9]: Of the above schemes, ACK-based protocols are intrinsically unsuited for time-critical multi-sender settings, while sender-based sequencing and FEC limit discovery latency to inter-send time at a single sender within a single group. Ideally, we would like discovery latency to be independent of inter-send time, and combine the scalability of a gossip-based scheme with the tunability of FEC. Receiver-based FEC, first introduced in the Slingshot protocol [9], provides such a combination: receivers generate FEC packets from incoming data and exchange these with other randomly chosen receivers. Since FEC packets are generated from *incoming* data at a receiver, the timeliness of loss recovery depends on the rate of data multicast in *the entire group*, rather than the rate at any given sender, allowing scalability in the number of senders to the group.

Slingshot is aimed at single-group settings, recovering from packet loss in time proportional to that group's data rate. Our goal with Ricochet is to achieve recovery latency dependent on the rate of data incoming at a node *across all groups*. Essentially, we want recovery of packets to occur as quickly in a thousand 10 Kbps groups as in a single 10 Mbps group, allowing applications to divide node bandwidth among thousands of multicast groups while maintaining time-critical packet recovery. To achieve this, we introduce Lateral Error Correction, a new form of receiver-generated FEC that probabilistically combines receiver-generated repair traffic across multiple groups to drive down packet recovery latencies.

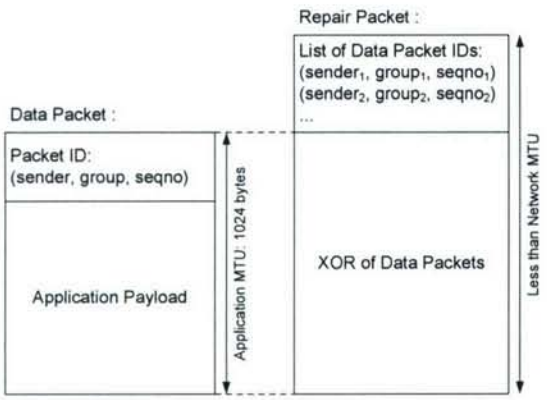


Figure 3: Ricochet Packet Structure

4 Lateral Error Correction and the Ricochet protocol

In Ricochet, each node belongs to a number of groups, and receives data multicast within any of them. The basic operation of the protocol involves generating XORs from incoming data and exchanging them with other randomly selected nodes. Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 3 shows the structure of these two packet types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from r data packets allows recovery of one of them, if all the other $r-1$ data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets.

At the core of Ricochet is the LEC engine running at each node that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind LEC is the notion that repair packets sent by a node to another node can be composed from data in any of the multicast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur within time that's inversely proportional to the aggregate rate of data in all these groups. Figure 4 illustrates this idea: n_1 has groups A and B in common with n_2, and hence it can generate and dispatch repair packets that contain data from both these groups. n_1 needs to wait only until it receives 5 data packets in either A *or* B before it sends a repair packet, allowing faster recovery of lost packets at n_2.

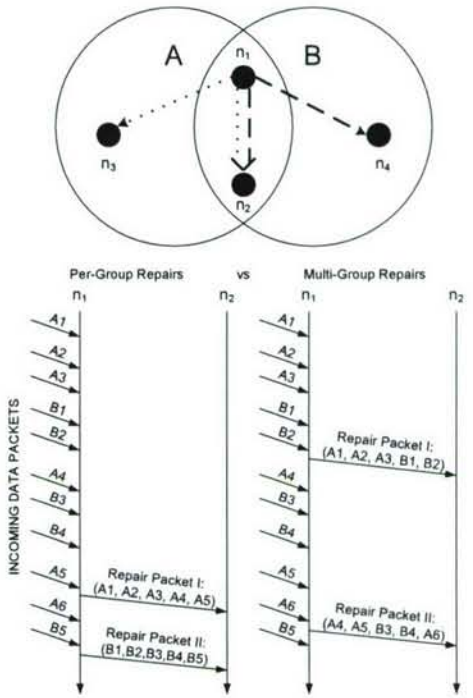


Figure 4: LEC in 2 Groups: Receiver n_1 can send repairs to n_2 that combine data from both groups A and B.

While combining data from different groups in outgoing repair packets drives down recovery time, it tampers with the coherent tunability that single group receiver-based FEC provides. The *rate-of-fire* parameter in receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every r data packets, c repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for LEC is to combine repair traffic for multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. Coherent, Tunable Per-Group Overhead: For every data packet that a node receives in a group with rate-of-fire (r, c), it sends out *an average of* c *repair packets* including that data packet to other nodes in the group.
2. Randomness: Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

LEC supports overlapping groups with the same r component and different c values in their rate-of-fire parameter. In LEC, the rate-of-fire parameter is translated into the following guarantee: For every data packet d that a node receives in a group with rate-of-fire (r, c), it selects an average of c nodes from the group randomly and sends each

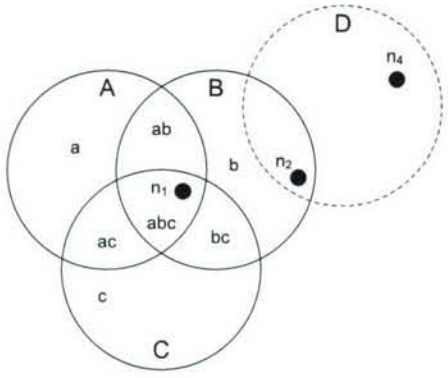


Figure 5: n_1 belongs to groups A, B, C: it divides them into disjoint regions abc, ab, ac, bc, a, b, c

of these nodes exactly one repair packet that includes d. In other words, the node sends an average of c repair packets containing d to the group. In the following section, we describe the algorithm that LEC uses to compose and dispatch repair packets while maintaining this guarantee.

4.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same LEC algorithm and supporting code runs at every node - and hence, we can describe its operation from the vantage point of a single node, n_1.

4.1.1 Regions

The LEC engine running at n_1 divides n_1's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that n_1 belongs to. Figure 5 shows the regions in a hypothetical system, where n_1 is in three groups, A, B and C. We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e, abc is a region formed by the intersection of A, B and C. In our example, the neighborhood set of n_1 is carved into seven regions: abc, ac, ab, bc, a, b and c, essentially the power set of the set of groups involved. Readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case; we are only concerned with non-empty intersections, the cardinality of which is bounded by the number of nodes in the system, as each node belongs to exactly one intersection (see Section 4.1.4). Note that n_1 does not belong to group D and is oblivious to it; it observes n_2 as belonging to region b, rather than bd, and is not aware of n_4's existence.

4.1.2 Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, LEC selects targets randomly from *each region*. The number of targets selected from a region is set such that:

1. It is proportional to the size of the region
2. The total number of targets selected, across regions, is equal to the c value of the group

Hence, for a given group A with rate-of-fire (r, c_A), the number of targets selected by LEC in a particular region, say abc, is equal to $c_A * \frac{|abc|}{|A|}$, where $|x|$ is the number of nodes in the region or group x. We denote the number of targets selected by LEC in region abc for packets in group A as c_A^{abc}. Figure 6 shows n_1 selecting targets for repairs from the regions of A.

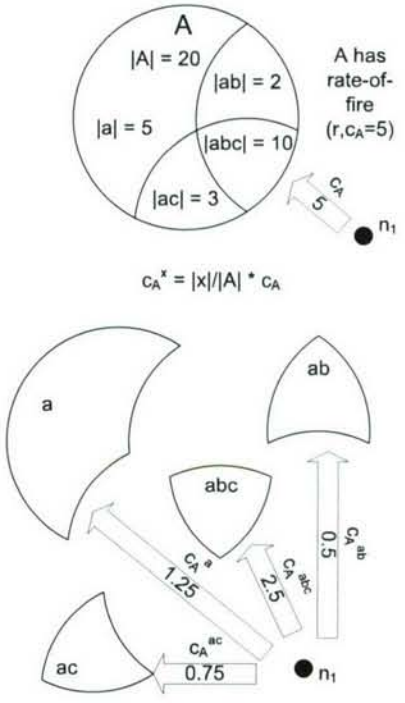


Figure 6: n_1 selects proportionally sized chunks of c_A from the regions of A

Note that LEC may pick a different number of targets from a region for packets in a different group; for example, c_A^{abc} differs from c_B^{abc}. Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size c has to be selected from the group, selecting targets from regions corresponds to *stratified sampling* [14], a technique from statistical theory.

4.1.3 Why select targets from regions?

Selecting targets from regions instead of groups allows LEC to construct repair packets from multiple groups; since we know that all nodes in region ab are interested in data from groups A and B, we can create composite

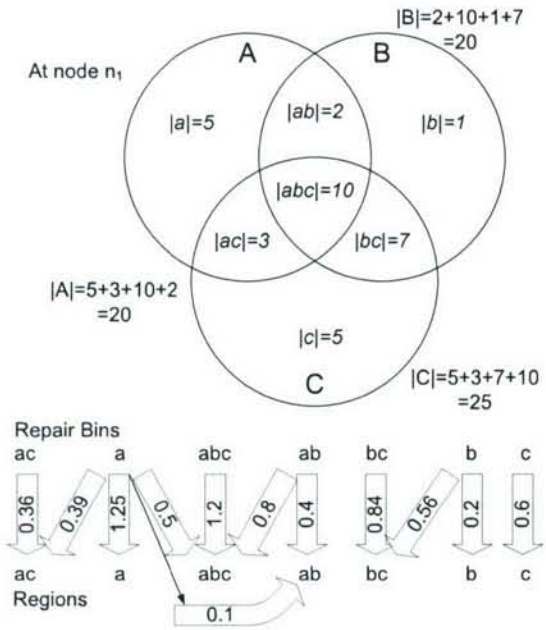


Figure 7: Mappings between repair bins and regions: the repair bin for ab selects 0.4 targets from region ab and 0.8 from abc for every repair packet. Here, $c_A = 5$, $c_B = 4$, and $c_C = 3$.

repair packets from incoming data packets in both groups and send them to nodes in that region.

Single-group receiver-based FEC [9] is implemented using a simple construct called a *repair bin*, which collects incoming data within the group. When a repair bin reaches a threshold size of r, a repair packet is generated from its contents and sent to c randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems simple; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size r, it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups A, B, and C, how many targets should we select from region abc for this repair packet - c_A^{abc}, c_B^{abc}, or c_C^{abc}? One possible solution is to pick the maximum of these values. If $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$, then we would select c_A^{abc}. However, a data packet in group B, when added to the repair bin for the region abc would be sent to an average of c_A^{abc} targets in the region; resulting in more repair packets containing that data packet sent to the region than required (c_B^{abc}), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group B than required by its (r, c_B) value; a similar argument holds for data packets in group C as well.

Algorithm 1 Algorithm for Setting Up Repair Bins

1: **Code at node n_i:**

2: **upon** Change in Group Membership **do**
3: **while** L not empty {*L is the list of regions*} **do**
4: Select and remove the region $R_i = abc...z$ from L with highest number of groups involved (break ties in any order)
5: Set $R_t = R_i$
6: **while** $R_t \neq \epsilon$ **do**
7: set c_{min} to $min(c_A^{R_t}, c_B^{R_t} ...)$, where {A,B,...} is the set of groups forming R_t
8: Set number of targets selected by R_i's repair bin from region R_t to c_{min}
9: Remove G from R_t, for all groups G where $c_G^{R_t} = c_{min}$
10: For each remaining group G' in R_t, set $c_{G'}^{R_t} = c_{G'}^{R_t} - c_{min}$

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups A and B than required, resulting in a lower fraction of packets recovered from LEC. To rectify this we send the additional compensating repair packets to the region abc from the repair bins for regions a and b. The repair bin for region a would select $c_A^{abc} - c_C^{abc}$ destinations, on an average, for every repair packet it generates; this is in addition to the c_A^a destinations it selects from region a.

A more sophisticated version of the above strategy involves iteratively obtaining the required repair packets from regions involving the remaining groups; for instance, we would have the repair bin for ab select the minimum of c_A^{abc} and c_B^{abc} - which happens to be c_B^{abc} - from abc, and then have the repair bin for a select the remainder value, $c_A^{abc} - c_B^{abc}$, from abc. Algorithm 1 illustrates the final approach adopted by LEC, and Figure 7 shows the output of this algorithm for an example scenario. A repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

4.1.4 Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that n_1 is part of. The algorithm has complexity $O(I \cdot d)$, where I is the number of populated regions (i.e, with one or more nodes in them), and d is the maximum number of groups that form a region. Note that I at n_1 is bounded from above by the cardinality of the set of nodes that share a multicast

group with n_1, since regions are disjoint and each node exists in exactly one of them. d is bounded by the number of groups that n_1 belongs to.

4.2 Implementation Details

Our implementation of Ricochet is in Java. Below, we discuss the details of the implementation, along with the performance optimizations involved - some obvious and others subtle.

4.2.1 Repair Bins

A Ricochet repair bin is a lightweight structure holding an XOR and a list of data packets, and supporting an *add* operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in Figure 7, the repair bin for *abc* has to select 1.2 targets from *abc*, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

4.2.2 Staggering for Bursty Loss

A crucial algorithmic optimization in Ricochet is *staggering* - also known as interleaving [23] - which provides resilience to bursty loss. Given a sequence of data packets to encode, a stagger of 2 would entail constructing one repair packet from the 1st, 3rd, 5th... packets, and another repair packet from the 2nd, 4th, 6th... packets. The stagger value defines the number of repairs simultaneously being constructed, as well as the distance in the sequence between two data packets included in the same repair packet. Consequently, a stagger of i allows us to tolerate a loss burst of size i while resulting in a proportional slowdown in recovery latency, since we now have to wait for $O(i*r)$ data packets before despatching repair packets.

In conventional sender-based FEC, staggering is not a very attractive option, providing tolerance to very small bursts at the cost of multiplying the already prohibitive loss discovery latency. However, LEC recovers packets so quickly that we can tolerate a slowdown of a factor of ten without leaving the tens of milliseconds range; additionally, a small stagger at the sender allows us to tolerate very large bursts of lost packets at the receiver, especially since the burst is dissipated among multiple groups and senders. Ricochet implements a stagger of i by the simple expedient of duplicating each logical repair bin into i instances; when a data packet is added to the logical repair bin, it is actually added to a particular instance of the repair bin, chosen in round-robin fashion. Instances of a duplicated repair bin behave exactly as single repair bins do, generating repair packets and sending them to regions when they get filled up.

4.2.3 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [24]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the LEC algorithm. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at n_1 - and consequently the group and intersection sizes - does not include n_1 itself.

4.2.4 Membership and Failure Detection

Ricochet can plug into any existing membership and failure detection infrastructure, as long as it is provided with reasonably up-to-date views of per-group membership by some external service. In our implementation, we use simple versions of Group Membership (GMS) and Failure Detection (FD) services, which execute on high-end server machines. If the GMS receives a notification from the FD that a node has failed, or it receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s). The GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node n_{55} joins group A, the update sent by the GMS to every node in A would be a 3-tuple: *(Join, A, n_{55})*. Individual nodes process these updates to construct multi-group views relative to their own membership.

Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this can be easily done by partitioning the service on group id and running each partition on a different server. For instance, one machine is responsible for groups A and B, another for C and D, and so on. Similarly, the FD can be partitioned on a topological criterion; one machine on each rack is responsible for monitoring other nodes on the rack by pinging them periodically. For fault-tolerance, each partition of the GMS can be replicated on multiple machines using a strongly consistent protocol like Paxos. The FD can have a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain. We believe that

such a semi-centralized solution is appropriate and sufficient in a datacenter setting, where connectivity and membership are typically stable. Crucially, the protocol itself does not need consistent membership, and degrades gracefully with the degree of inconsistency in the views; if a failed node is included in a view, performance will dip fractionally in all the groups it belongs to as the repairs sent to it by other nodes are wasted.

4.2.5 Performance

Since Ricochet creates LEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to: Instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to c (from the rate-of-fire) in the worst case - which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

4.2.6 Buffering and Loss Control

LEC - like any other form of FEC - works best when losses are not in concentrated bursts. Ricochet maintains an application-level buffer with the aim of minimizing in-kernel losses, serviced by a separate thread that continuously drains packets from the kernel. If memory at end-hosts is constrained and the application-level buffer is bounded, we use customized packet-drop policies to handle overflows: a randomly selected packet from the buffer is dropped and the new packet is accommodated instead. In practice, this results in a sequence of almost random losses from the buffer, which are easy to recover using FEC traffic. Whether the application-level buffer is bounded or not, it ensures that packet losses in the kernel are reduced to short bursts that occur only during periods of overload or CPU contention. We evaluate Ricochet against loss bursts of up to 100 packets, though in practice we expect the kind of loss pattern shown in 1, where few bursts are greater than 20-30 packets, even with highly concentrated traffic spikes.

4.2.7 NAK Layer for 100% Recovery

Ricochet recovers a high percentage of lost packets via the proactive LEC traffic; for certain applications, this probabilistic guarantee of packet recovery is sufficient and even desirable in cases where data 'expires' and there is no utility in recovering it after a certain number of milliseconds. However, the majority of applications require 100% recovery of lost data, and Ricochet uses a reactive NAK layer to provide this guarantee. If a receiver does not recover a packet through LEC traffic within a timeout period after discovery of loss, it sends an explicit NAK to the sender and requests a retransmission. While this NAK layer does result in extra reactive repair traffic, two factors separate it from traditional NAK mechanisms: firstly, recovery can potentially occur very quickly - within a few hundred milliseconds - since for almost all lost packets discovery of loss takes place within milliseconds through LEC traffic. Secondly, the NAK layer is meant solely as a backup mechanism for LEC and responsible for recovering a very small percentage of total loss, and hence the extra overhead is minimal.

4.2.8 Optimizations

Ricochet maintains a buffer of unusable repair packets that enable it to utilize incoming repair packets better. If one repair packet is missing exactly one more data packet than another repair packet, and both are missing at least one data packet, Ricochet obtains the extra data packet by XORing the two repair packets. Also, it maintains a list of unusable repair packets which is checked intermittently to see if recent data packet recoveries and receives have made any old repair packets usable.

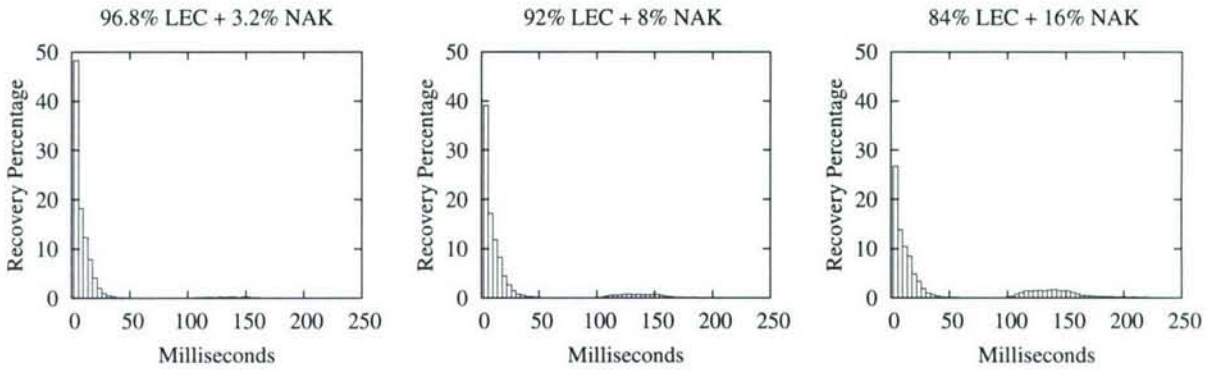
4.2.9 Message Ordering

As presented, Ricochet provides multicast reliability but does not deliver messages in the same order at all receivers. We are primarily concerned with building an extremely rapid multicast primitive that can be used by applications that require unordered reliable delivery as well as layered under ordering protocols with stronger delivery properties. For instance, Ricochet can be used as a reliable transport by any of the existing mechanisms for total ordering [16] — in separate work [8], we describe one such technique that predicts out-of-order delivery in datacenters to optimize ordering delays.

5 Evaluation

We evaluated our Java implementation of Ricochet on a 64-node cluster, comprising of four racks of 16 nodes each, interconnected via two levels of switches. Each node has a single 1.3 GHz CPU with 512 Mb RAM, runs Linux 2.6.12 and has two 100 Mbps network interfaces, one for control and the other for experimental traffic. Typical socket-to-socket latency within the cluster is around 50 microseconds. In the following experiments, for a given loss rate L, three different loss models are used:

· **uniform** - also known as the Bernoulli model [25] - refers to dropping packets with uniform probability equal to the loss rate L.



(a) 10% Loss Rate **(b) 15% Loss Rate** **(c) 20% Loss Rate**

Figure 8: Distribution of Recoveries: LEC + NAK for varying degrees of loss

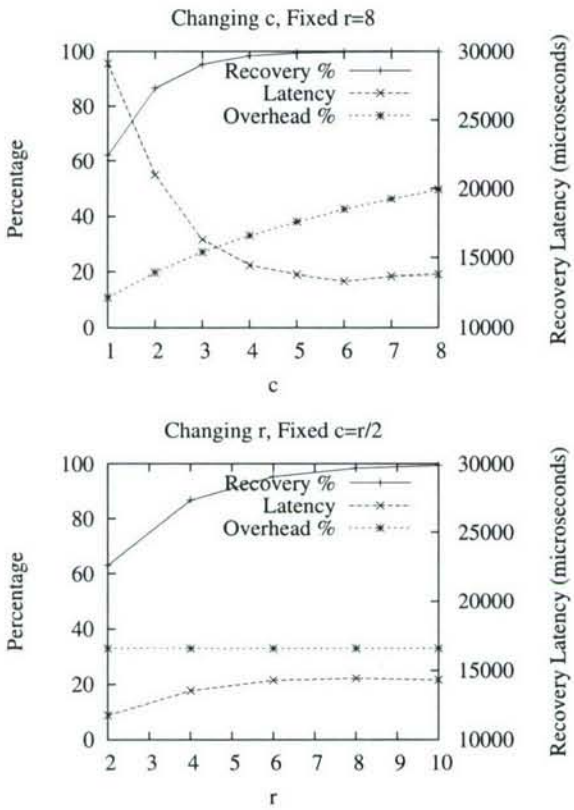


Figure 9: Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire (r, c).

· **bursty** involves dropping packets in equal bursts of length b. The probability of starting a loss burst is set so that each burst is of exactly b packets and the loss rate is maintained at L. This is not a realistic model but allows us to precisely measure performance relative to specific burst lengths.

· **markov** drops packets using a simple 2-state markov chain, where each node alternates between a lossy and a lossless state, and the probabilities are set so that the average length of a loss burst is m and the loss rate is L, as described in [25].

In experiments with multiple groups, nodes are assigned to groups at random, and the following formula is used to relate the variables in the grouping pattern: $n * d = g * s$, where n is the number of nodes in the system (64 in most of the experiments), d is the degree of membership, i.e. the number of groups each node joins, g is the total number of groups in the system, and s is the average size of each group. For example, in a 16-node setting where each node joins 512 groups and each group is of size 8, g is set to $\frac{16*512}{8} \approx 1024$. Each node is then assigned to 512 randomly picked groups out of 1024. Hence, the grouping patterns for each experiment is completely represented by a (n, d, s) tuple.

For every run, we set the sending rate at a node such that the total system rate of incoming messages is 64000 packets per second, or 1000 packets per node per second. Data packets are 1K bytes in size. Each point in the following graphs - other than Figure 8, which shows distributions for single runs - is an average of 5 runs. A run lasts 30 seconds and produces $\approx$ 2 million receive events in the system.

5.1 Distribution of Recoveries in Ricochet

First, we provide a snapshot of what typical packet recovery timelines look like in Ricochet. Earlier, we made

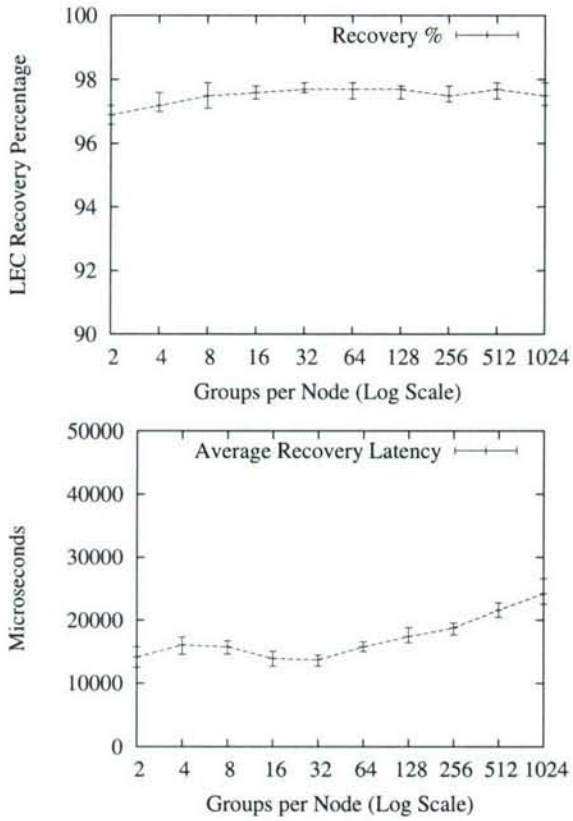


Figure 10: Scalability in Groups

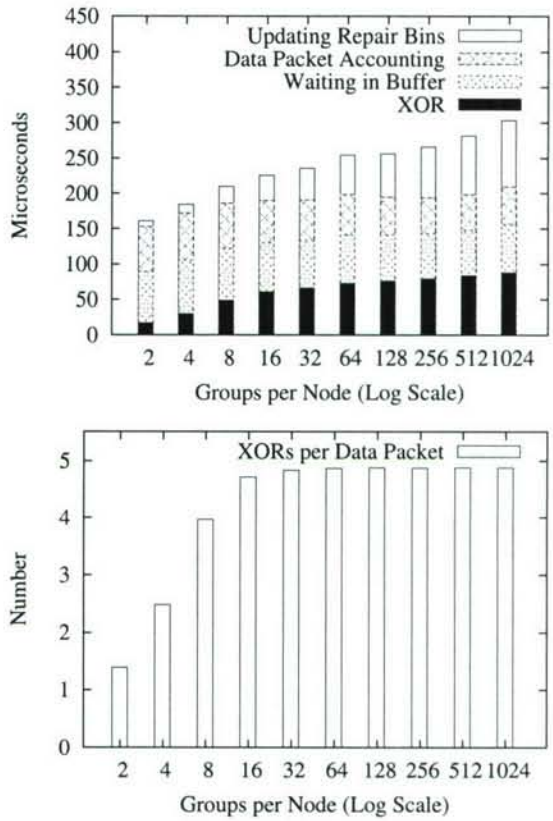
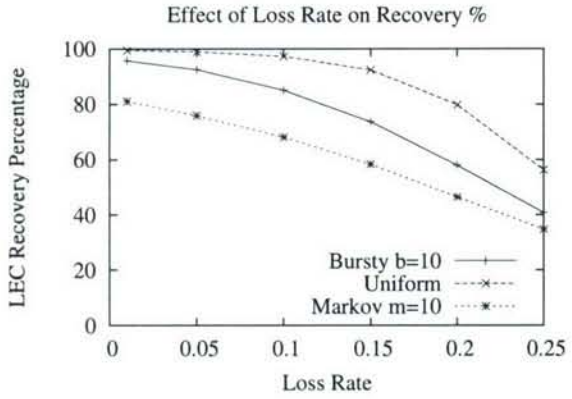


Figure 11: CPU time and XORs per data packet

the assertion that Ricochet discovers the loss of almost all packets very quickly through LEC traffic, recovers a majority of these instantly and recovers the remainder using an optional NAK layer. In Figure 8, we show the histogram of packet recovery latencies for a 16-node run with degree of membership $d = 128$ and group size $s = 10$. We use a simplistic NAK layer that starts unicasting NAKs to the original sender of the multicast 100 milliseconds after discovery of loss, and retries at 50 millisecond intervals. Figure 8 shows three scenarios: under uniform loss rates of 10%, 15%, and 20%, different fractions of packet loss are recovered through LEC and the remainder via reactive NAKs. These graphs illustrate the meaning of the LEC recovery percentage: if this number is high, more packets are recovered very quickly without extra traffic in the initial segment of the graphs, and less reactive overhead is induced by the NAK layer. Importantly, even with a recovery percentage as low as 84% in Figure 8(c), we are able to recover all packets within 250 milliseconds with a crude NAK layer due to early LEC-based discovery of loss. For the remaining experiments, we will switch the NAK layer off and focus solely on LEC performance; also, since we found this distribution of recovery latencies to be fairly representative, we present only the percentage of lost packets recovered using LEC and the average latency of these recoveries. Experiment Setup: $(n = 16, d = 128, s = 10)$, Loss Model: Uniform, [10%, 15%, 20%].

5.2 Tunability of LEC in multiple groups

The Slingshot protocol [9] illustrated the tunability of receiver-generated FEC for a single group; we include a similar graph for Ricochet in Figure 9, showing that the rate-of-fire parameter (r, c) provides a knob to tune LEC's recovery characteristics. In Figure 9.a, we can see that increasing the c value for constant $r = 8$ increases the recovery percentage and lowers recovery latency by expending more overhead - measured as the percentage of repair packets to all packets. In Figure 9.b, we see the impact of increasing r, keeping the ratio of c to r - and consequently, the overhead - constant. For the rest of the experiments, we set the rate-of-fire at $(r = 8, c = 5)$. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Uniform, 1%.



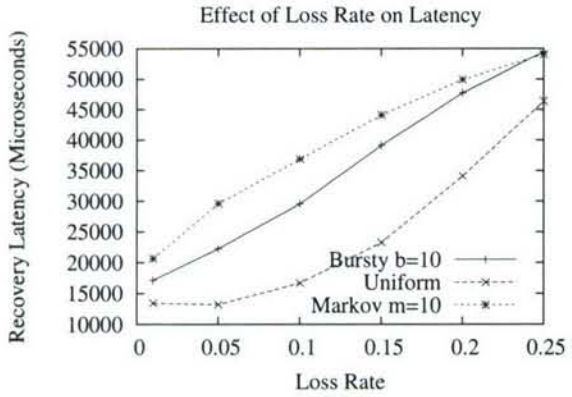
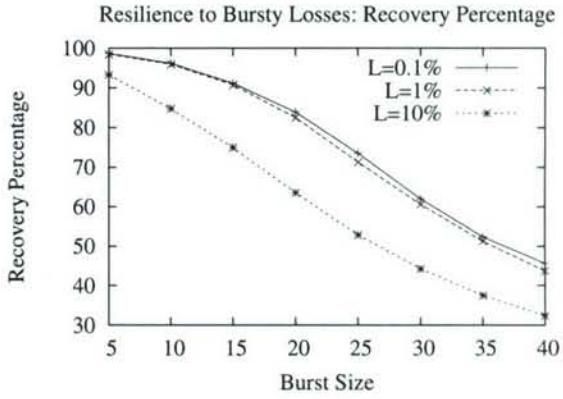


Figure 12: Impact of Loss Rate on LEC



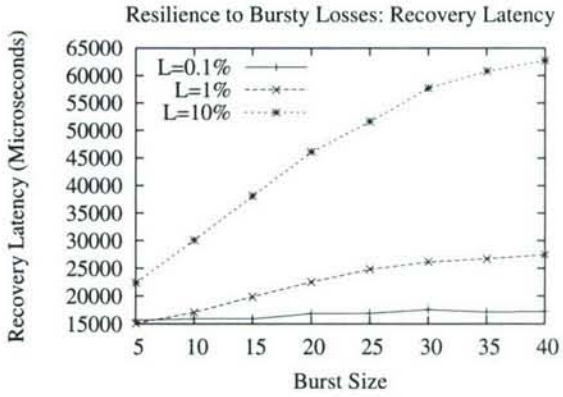


Figure 13: Resilience to Burstiness

5.3 Scalability

Next, we examine the scalability of Ricochet to large numbers of groups. Figure 10 shows that increasing the degree of membership for each node from 2 to 1024 has almost no effect on the percentage of packets recovered via LEC, and causes a slow increase in average recovery latency. The x-axis in these graphs is log-scale, and hence a straight line increase is actually logarithmic with respect to the number of groups and represents excellent scalability. The increase in recovery latency towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section.

For a comparison point, we refer readers back to SRM's discovery latency in Figure 2: in 128 groups, SRM discovery took place at 9 seconds. In our experiments, SRM recovery took place roughly 4 seconds after discovery in all cases. While fine-tuning the SRM implementation for clustered settings should eliminate that 4 second gap between discovery and recovery, at 128 groups Ricochet surpasses SRM's best possible recovery performance of 5 seconds by between 2 and 3 orders of magnitude.

Though Ricochet's recovery characteristics scale well in the number of groups, it is important that the computational overhead imposed by the protocol on nodes stays manageable, given that time-critical applications are expected to run over it. Figure 11 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a log x-axis shows that per-packet processing time increases logarithmically with the number of groups - doubling the number of groups results in a constant increase in processing time. The increase in processing time towards the latter half of the graph is due to the increase in the number of repair bins with the number of groups. While we considered 1024 groups adequate scalability, Ricochet can potentially scale to more groups with further optimization, such as creating bins only for occupied regions. In the current implementation, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second. Figure 11 also shows the average number of XORs per incoming data packet. As stated in section 4.2.2, the number of XORs stays under 5 - the value of c from the rate-of-fire (r, c). Experiment Setup: $(n = 64, d = *, s = 10)$, Loss Model: Uniform, 1%.

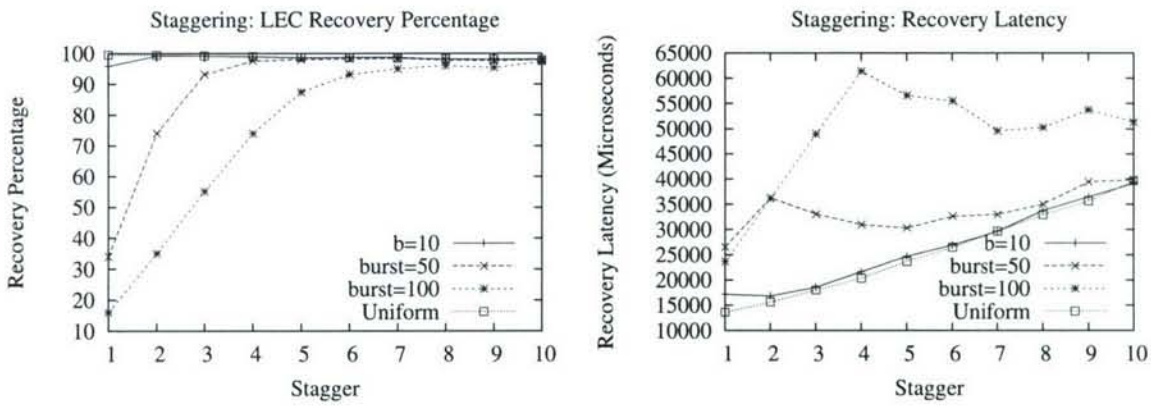


Figure 14: Staggering allows Ricochet to recover from long bursts of loss.

5.4 Loss Rate and LEC Effectiveness

Figure 12 shows the impact of the Loss Rate on LEC recovery characteristics, under the three loss models. Both LEC recovery percentages and latencies degrade gracefully: with an unrealistically high loss rate of 25%, Ricochet still recovers 40% of lost packets at an average of 60 milliseconds. For uniform and bursty loss models, recovery percentage stays above 90% with a 5% loss rate; markov does not fare as well, even at 1% loss rate, primarily because it induces bursts much longer than its average of 10 - the max burst in this setting averages at 50 packets. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: *.

5.5 Resilience to Bursty Losses

As we noted before, a major criticism of FEC schemes is their fragility in the face of bursty packet loss. Figure 13 shows that Ricochet is naturally resilient to small loss bursts, without the stagger optimization - however, as the burst size increases, the percentage of packets recovered using LEC degrades substantially. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty.

However, switching on the stagger optimization described in Section 4.2.2 increases Ricochet's resilience to burstiness tremendously, without impacting recovery latency much. Figure 14 shows that setting an appropriate stagger value allows Ricochet to handle large bursts of loss: for a burst size as large as 100, a stagger of 6 enables recovery of more than 90% lost packets at an average latency of around 50 milliseconds. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty, 1%.

5.6 Effect of Group and System Size

What happens to LEC performance when the average group size in the cluster is large compared to the total number of nodes? Figure 15 shows that recovery percentages are almost unaffected, staying above 99% in this scenario, but recovery latency is impacted by more than a factor of 2 as we triple group size from 16 to 48 in a 64-node setting. Note that this measures the impact of the size of the group relative to the entire system; receiver-based FEC has been shown to scale well in a single isolated group to hundreds of nodes [9]. Experiment Setup: $(n = 64, d = 128, s = *)$, Loss Model: Uniform, 1%.

While we could not evaluate to system sizes beyond 64 nodes, Ricochet should be oblivious to the size of the entire system, since each node is only concerned with the groups it belongs to. We ran 4 instances of Ricochet on each node to obtain an emulated 256 node system with each instance in 128 groups, and the resulting recovery percentage of 98% - albeit with a degraded average recovery latency of nearly 200 milliseconds due to network and CPU contention - confirmed our intuition of the protocol's fundamental insensitivity to system size.

6 Future Work

One avenue of research involves embedding more complex error codes such as Tornado [11] in LEC; however, the use of XOR has significant implications for the design of the algorithm, and using a different encoding might require significant changes. LEC uses XOR for its simplicity and speed, and as our evaluation showed, we obtain properties on par with more sophisticated encodings, including tunability and burst resilience. We plan on replacing our simplistic NAK layer with a version optimized for bulk transfer, providing an efficient backup for LEC when sustained bursts occur of hundreds of packets or more. Another line of work involves making the parameters for LEC - such as rate-of-fire and stagger - adaptive, reacting to meet varying load and network characteristics. We are currently working with industry partners to layer Ricochet under data distribution, publish-subscribe and web-service interfaces, as well as building protocols with stronger ordering and atomicity properties over it.

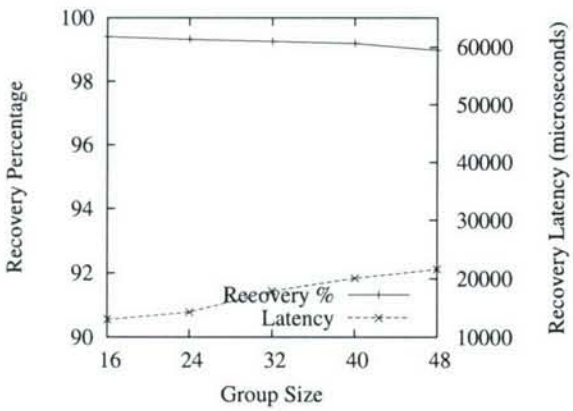


Figure 15: Effect of Group Size

7 Conclusion

We believe that the next generation of time-critical applications will execute on commodity clusters, using the techniques of massive redundancy, fault-tolerance and scalable communication currently available to distributed systems practitioners. Such applications will require a multicast primitive that delivers data at the speed of hardware multicast in failure-free operation and recovers from packet loss within milliseconds irrespective of the pattern of usage. Ricochet provides applications with massive scalability in multiple dimensions - crucially, it scales in the number of groups in the system, performing well under arbitrary grouping patterns and overlaps. A clustered communication primitive with good timing properties can ultimately be of use to applications in diverse domains not normally considered time-critical - e-tailers, online webservers and enterprise applications, to name a few.

Acknowledgments

We received invaluable comments from Dave Andersen, Danny Dolev, Tudor Marian, Art Munson, Robbert van Renesse, Emin Gun Sirer, Niraj Tolia and Einar Vollset. We would like to thank our shepherd Mike Dahlin, as well as all the anonymous reviewers of the paper.

References

[1] Bea weblogic. `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic`, 2006.

[2] Gemstone gemfire. `http://www.gemstone.com/products/gemfire/enterprise.php`, 2006.

[3] Ibm websphere. `www.ibm.com/software/webservers/appserv/was/`, 2006.

[4] Jboss. `http://labs.jboss.com/portal/`, 2006.

[5] Real-time innovations data distribution service. `http://www.rti.com/products/data_distribution/index.html`, 2006.

[6] Tangosol coherence. `http://www.tangosol.com/html/coherence-overview.shtml`, 2006.

[7] Tibco rendezvous. `http://www.tibco.com/software/messaging/rendezvous.jsp`, 2006.

[8] M. Balakrishnan, K. Birman, and A. Phanishayee. Plato: Predictive latency-aware total ordering. In *IEEE SRDS*, 2006.

[9] M. Balakrishnan, S. Pleisch, and K. Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*, 2005.

[10] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98 Conference*, pages 56–67, New York, NY, USA, 1998. ACM Press.

[12] Y. Chawathe, S. McCanne, and E. A. Brewer. Rmx: Reliable multicast for heterogeneous networks. In *INFOCOM*, pages 795–804, 2000.

[13] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *ACM SIGCOMM*, pages 55–67, New York, NY, USA, 2001. ACM Press.

[14] W. G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.

[15] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[17] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[18] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[19] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The pgm reliable multicast protocol. *IEEE Network*, 17(1):16–22, Jan 2003.

[20] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[21] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[22] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, Apr. 1998. IETF Internet Draft.

[23] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.

[24] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[25] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *INFOCOM*, pages 345–352, 1999.

MISTRAL: Efficient Flooding in Mobile Ad-hoc Networks*

Stefan Pleisch† Mahesh Balakrishnan‡ Ken Birman‡ Robbert van Renesse‡

†Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland

‡Department of Computer Science
Cornell University, Ithaca, NY 14853, USA

stefan.pleisch@epfl.ch {mahesh|ken|rvr}@cs.cornell.edu

ABSTRACT

Flooding is an important communication primitive in mobile ad-hoc networks and also serves as a building block for more complex protocols such as routing protocols. In this paper, we propose a novel approach to flooding, which relies on proactive compensation packets periodically broadcast by every node. The compensation packets are constructed from dropped data packets, based on techniques borrowed from forward error correction. Since our approach does not rely on proactive neighbor discovery and network overlays it is resilient to mobility.

We evaluate the implementation of Mistral through simulation and compare its performance and overhead to purely probabilistic flooding. Our results show that Mistral achieves a significantly higher node coverage with comparable overhead.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*;
C.4 [**Computer-Communication Networks**]: Performance of Systems—*Reliability, availability, and serviceability*;
C.4 [**Computer-Communication Networks**]: Performance of Systems—*Fault tolerance*

General Terms

Algorithms, reliability

Keywords

Mobile ad hoc networks, MANET, flooding, forward error correction, compensation

*Our effort is supported by the Swiss National Science Foundation (SNF), NSF Trust STC, the NSP NetNOSS program, and the DARPA ACERT program.

MobiHoc'06, May 22–25, 2006, Florence, Italy.

1. INTRODUCTION

Mobile ad hoc networks (MANETs) have received much attention in recent years. A MANET is a multi-hop wireless network without fixed infrastructure, in which nodes can be mobile. MANETs are increasingly important because wireless communication is rapidly becoming ubiquitous. Potential applications range from military and disaster response applications to more traditional urban problems such as finding desired products or services in a city. The devices themselves are diverse, including PDAs, cell phones, sensors, laptops, etc. Many new protocols have been proposed to solve the technical problems confronted in MANETs and to offer platform support for applications that collect and exploit the data available in such settings.

Because of the lack of a fixed communication infrastructure, flooding in MANETs [10] is an important communication primitive and also serves as a building block for more complex protocols such as AODV [21] or ODMRP [16]. *Flooding* is the mechanism by which a node, receiving flooded message m for the first time, rebroadcasts m once. We distinguish between flooding and *broadcast*, which is a transmission that is received by all nodes within transmission range of the broadcasting node. Flooding usually covers all the nodes in a network, but can also be limitted to a set of nodes that is defined by a geographical area (also called *geocast flooding* [14]) or by the time-to-live (TTL) parameter of m. Thus, a node receiving the flooded message only rebroadcasts it if it is within the specified area or if the message's TTL is greater than 0.

Unfortunately, flooding has been shown to be susceptible to contention even in reasonably dense networks [18]. Indeed, flooding leads to a large amount of redundant messages that consume scarce resources such as bandwidth and power and cause contention, collisions and thus additional packet loss. Every node receives the message from every neighbor within transmission range, except when messages are lost due to contention and collisions. This problem is known as the *broadcast storm* problem [18]. Because flooding is important in MANET applications, there is a clear need for storm-resistant flooding protocols that operate efficiently. However, reducing the number of redundant broadcasts leads to a lower degree of reliability. Hence, the challenge we face is to strike a balance between message overhead (i.e., the level of redundancy) and reliability.

To reduce the number of redundant messages, two ba-

sic classes of mechanisms have been proposed: (1) imposing a (partial) routing overlay structure; and (2) selectively dropping messages. Approaches in (1) build and maintain a (partial) routing overlay structure in the ad hoc network, which is used to efficiently broadcast the flooded message. For instance, only nodes that are part of a multicast tree rebroadcast the message [20]. Other approaches in this category are [3, 8, 17]. With mobile nodes the underlying routing structure needs to be frequently changed, incurring high maintenance costs and generally reduced reliability during the restructuring. In contrast, approaches in (2) do not rely on an explicit underlying routing structure. Instead, each node uses local information to make an independent decision whether to rebroadcast or to drop the flooded message. The simplest approach in this class is purely probabilistic flooding [18], in which messages are rebroadcast with a certain fixed probability. While probabilistic flooding reduces the number of broadcasts, when applied naively it simply recreates our earlier problem: poorly connected nodes (those with few neighbors) may fail to receive a flooded message. This consideration has motivated a number of more complex approaches, such as the algorithms given in [28, 24].

In our paper, we focus on class (2) but propose a new mechanism to reduce the number of missed flooded messages. We start with purely probabilistic flooding [18] but compensate for dropped data packets by periodically broadcasting *compensation packets*. Every compensation packet encodes a set of packets that have been dropped (i.e., that are not rebroadcast) by the sender. A node's neighbors, upon receipt of such a packet, can recover missing packets if it already has received and buffered a sufficient percentage of the packets that were used in constructing the compensation packet.

Even when a node has lost too many packets to reconstruct missing data, the compensation packets provide information that can be used to identify the loss. We include a secondary recovery mechanism that kicks in when a node discovers an unrecoverable loss, and part of our task in the evaluation presented here is to quantify the tradeoff between the additional message overhead versus increased reliability.

We have implemented Mistral and simulate its performance on JiST/SWANS, a simulation package that lets the developer run code in an emulated environment. Our results show that compensation packets significantly increase coverage when compared to probabilistic flooding with comparable overhead.

The remainder of the paper is structured as follows: Section 2 overviews the problem of flooding and places our work in the context of earlier work. In Section 3 we introduce the Mistral algorithm. Section 4 provides a simple analysis of Mistral. In Section 5, we present the simluation results and measure Mistral's performance. We conclude the paper with Section 6.

2. FLOODING IN MANETS

In any flooding mechanism, one must balance reliability against message overhead. On the one hand, increasing reliability generally involves sending a greater number of redundant messages and thus incurs a higher message overhead. In this worst case, the system risks provoking broadcast storms. Yet redundant messages are needed to reach all nodes and to recover from packet loss, hence reducing the overhead will generally decrease reliability.

The broadcast storm problem is so common in flooding algorithms that it has engendered a whole area of research. Storm-sensitive flooding approaches can be broadly classified into two classes: *local-knowledge-based* and *overlay-based*. Local-knowledge-based approaches decide on whether to rebroadcast or drop a flooded message solely on the basis of local information. Most commonly, they use information from received broadcasts to adaptively determine the forwarding policy. Such algorithms are a natural fit for MANETs, as they do not need to maintain any kind of complex node-to-node state that might need to be adapted in the event of mobility or other topology changes. In contrast, overlay-based approaches structure the node field according to some (local) topology, and then use topological information to efficiently implement flooding and reliability. The problem here is that if nodes have low quality connections to neighbors and/or are in motion, the overlay structure must be adapted. As a consequence, a high rate of management messages may be required, and if a flooded message is propagated while the overlay is out of date, that message may experience a high loss rate. In the worst case, the system might end up in a state of churn, constantly adapting the overlay but never managing to achieve the high quality of flooding that the overlay is intended to support.

We now briefly overview existing work and assign it to the corresponding class. For reasons of brevity, our review is deliberately partial; we focus on results that inspired our work here, or that have been widely cited in the literature. For a more comprehensive overview that includes a comparison of some of the major flooding approaches the reader is referred to [26].

2.1 Overlay-Based Approaches

As just indicated, we use the term *overlay* very broadly. For us, an overlay-based approaches is an algorithm that superimposes a routing structure onto the ad hoc network in support of flooding and rebroadcast. Depending on the position of a node in this overlay, it decides to either rebroadcast a flooded packet, or to only process and then drop it. While overlays provide a convenient mechanism to reduce the message overhead of flooding and to increase reliability, they suffer from the need to reconfigure the overlay when connectivity changes or if the nodes are mobile. Restructuring adds overhead but also increases the likelihood that messages will be lost, and thus may decrease coverage of the flooding protocol.

Ni *et al.* [18] propose to structure the nodes into clusters. Their solution rebroadcasts a packet in a manner that depends on the node's position in the cluster: only cluster head and gateway nodes rebroadcast.

In [8], the goal is to provide low-latency flooding. This is in part achieved by minimizing the collisions and interference. Gandi *et al.* show that an optimal solution to this problem is NP complete, instead, they propose an approximation algorithm. They construct a multicast tree and compute a rebroadcasting schedule such that the expected rate of collisions will be low.

Other approaches are based on the approximation of (minimal) connected dominating sets (MCDS), e.g., [5] [3]. Informally, a dominating set (DS) contains a subset of all nodes such that every node not in the DS is adjacent to one in the DS. Thus, a DS creates a virtual backbone that can be used to efficiently flood messages. It has been shown that

the creation of an MCDS is NP-complete. Thus, most approaches attempt to find a sufficiently good approximation to a MCDS.

A number of approaches rely on two-hop neighbor information to select nodes that rebroadcast the message. These approaches require that *hello* messages containing neighbor information are exchanged between the nodes.

For instance, in the Double-Covered Broadcast (DCB) [17], node n collects information about the two-hop neighbor set. Among its one-hop neighbors it then picks nodes that rebroadcast the message (called *forward node*) such that (1) the rebroadcast by the forward node covers the two-hop neighbors, and (2) the one-hop neighbors that are no forward nodes are within range of at least two rebroadcasts by forward nodes. The reception of the message by the forward node is implicitly acknowledged when n overhears the rebroadcast.

The scalable broadcast algorithm (SBA) [20] also uses two-hop neighbor knowledge, but employs a different approach to select the forward nodes.

With node mobility, the two-hop neighbor sets need to be updated frequently. Otherwise, the neighbor sets become outdated and reliability drops (as observed in [17]).

2.2 Local-Knowledge-Based Approaches

Local-knowledge-based approaches generally decide on a per-node basis whether to rebroadcast a particular flooded message. In the simplest case, each node flips a coin and rebroadcasts messages with a certain probability p [18]. We call this approach *purely probabilistic flooding (PPF)*.

There are a number of variants on this basic idea. For example, one set of algorithms base the rebroadcast decision either on the number of already overheard rebroadcasts, or on the distance or location of the overheard rebroadcast's sender [18]. The idea underlying these schemes is that the additional coverage gained by rebroadcasting decreases with the number of overheard rebroadcasts and decreasing distance to neighboring rebroadcasting nodes. However, it takes time to collect these statistics, delaying the rebroadcast decision, hence a potentially high latency is introduced to every flooded message. In [25], Tseng *et al.* extend earlier approaches in [18] to allow nodes to dynamically adapt threshold values such as the rebroadcast counter.

In [28] Zhang and Agrawal propose an approach that is a combination of the counter-based and probabilistic method of [18]. Instead of using a static rebroadcast probability p, they adjust p according to the information collected by the counters. While this makes p adaptable, it becomes dependent upon other fixed parameters that need to be carefully selected (e.g., timeouts).

Dynamic Gossip [24] relies on local density awareness to adjust the rebroadcast probability p of the one-hop neighbors. Its correctness and suitability relies on the assumption that the nodes are uniformly distributed. Density information is collected using a relay-ping method.

In [15], Kowalski and Pelc propose a broadcasting algorithm with optimal lower bounds in their model. They consider only stationary nodes and ajust the broadcast probability accordingly.

Haas *et al.* [9] study what they term a *phase transition phenomena*. This work shows that purely probabilistic flooding (called *gossiping* in [9]) in an ad hoc network has a *bimodal* delivery distribution. Their simulations reveal that either almost every node receives the message, or virtually none. To reduce the likelihood of the latter case, they explore a variety of approaches, such as adapting the rebroadcast probability to the density or the distance to the flooding source. Sasson *et al.* [23] theoretically explore the same phenomena based on percolation theory and conclude that there exists a threshold $\bar{p} < 1$ such that for any $p > \bar{p}$ the node coverage is close to 1, while for $p < \bar{p}$ the coverage is very low. Hence, increasing p much beyond $\bar{p}$ is not very useful.

Any approach that bases rebroadcast decision on observation of neighbors and on overheard broadcasts is at risk of using stale information if nodes might move before the information is used. MANETs, of course, can have a high degree of mobility, hence neither of these approaches is ideal.

Mistral's compensation mechanisms is orthogonal to these approaches. Indeed, were we building a production deployment of flooding in a real-world setting, we would be inclined to combine Mistral with one of these others (as should be clear, the ideal choice of underlying mechanism depends upon the anticipated density of nodes and level of mobility; no single solution stands out as uniformly superior to the others). By using such a hybrid scheme, we could parameterize the underlying solution to keep overheads low, accepting a modest risk that flooded packets would fail to reach some nodes. Compensation packets could then be used to overcome this low level of residual losses.

3. MISTRAL

Traditional flooding suffers from the problem of redundant message reception, once per neighbor. Even in a reasonably connected network, the same message is received multiple times by every node, which is inefficient, wastes valuable resources, and can create contention in the transmission medium.

Selective rebroadcasting of flooded messages is a way to limit the number of redundant transmissions. Instead of simply rebroadcasting the message a node evaluates a local function $\mathcal{F}$ and then uses the outcome of this computation to decide whether to forward the message. In its simplest form, this function returns its result based on some static probability (corresponding to PPF). More complex functions take into account additional topological (e.g., the number of neighbors) or statistical information (e.g., the number of overheard rebroadcasts). The downside of selective flooding is that a flooding may no longer reach all intended nodes. In particular, if a node has only few neighbors, none of these neighbors may rebroadcast the message. Selective flooding thus balances message overhead against reliability.

Mistral finds some middle ground by introducing a new mechanism that allows us to fine-tune the balance between message overhead and reliability. The key idea is to extend selective flooding approaches by compensating for messages that are not rebroadcast. This compensation is based on a technique borrowed from forward error correction (FEC). Every incoming data packet (dp) is either rebroadcast or added to a compensation packet (cp). The compensation packet is broadcast at regular intervals and allows the receivers to recover one missing data packet.

3.1 Forward Error Correction

In its simplest form, Forward Error Correction (FEC) [11,

19, 22] creates l repair packets for every m data packets such that any m out of the resulting $(m+l)$ packets is enough to recover the original m data packets [11]. Traditional applications of FEC generate l repair packets for every m data packets and inject them into a data stream, which insulates the receiver from at most l packet losses. One of the fundamental advantages of FEC is that it imposes a constant overhead on the system and has easily understandable behavior under arbitrary network conditions. However, this simple form of FEC was developed for streaming settings, where a single sender is transmitting data at a high, steady rate such as in bulk file transfers [6] or in a video or audio feeds [7]. Part of our challenge is to develop a FEC solution matched to the characteristics of a MANET.

3.2 Algorithm

We noted earlier that Mistral can be built on top of any local-knowledge-based flooding approach. In the current implementation of the system, we use purely probabilistic flooding, mostly because this approach is extremely simple and is intuitively easy to visualize. Recall that in PPF, a node rebroadcasts a flooded message with static probability p. Although PPF might not be an ideal choice of algorithm in a practical deployment, the algorithm has no "hidden" effects that might make it hard to interpret our experimental findings.

Upon reception of a data packet, every node evaluates the function $\mathcal{F} : dp \longmapsto \{true|false\}$. In its most basic form, $\mathcal{F}$ takes a data packet as input and returns a boolean. If it returns true, dp is rebroadcast; otherwise, dp is added to the current compensation packet. When the number of data packets contained in a compensation packet passes a certain threshold c, the compensation packet is broadcast. We call c the *compensation rate*. Thus, a compensation packet is broadcast for every c data packets that are not rebroadcast.

Algorithm 1 presents the algorithm in more detail: Procedure `process` delivers the data packet to the application and decides whether to rebroadcast the packet or add it to the compensation packet; `composeCompensationPac` builds the compensation packet; and `runRecovery` attempts to recover data packets from stored compensation packets when a new data packet is delivered to the application. Finally, procedure `expand` is used for level-2 recovery, which is presented in Section 3.2.2. The secondary recovery mechanism discussed in the introduction is not included in Algorithm 1.

3.2.1 Composition of a Compensation Packet

In this section, we assume that data packets are of fixed size, e.g., 512 bytes, and contain the payload, a sender ID and some locally unique sequence number; we call these the *packet id*. The payload is assumed to remain unchanged during the course of the flooding (in some protocols, payloads do change as packets are routed; we discuss the handling of this kind of mutable payloads later in the paper).

To encode the payload of the data packets into the compensation packet, we use the XOR (operator $\otimes$), which is the simplest and best known FEC mechanism. A new data packet is added to the compensation packet by computing the XOR of its payload with the current payload in the compensation packet (initially, zero). Obviously, much more sophisticated error correction mechanisms are also possible; the advantage of XOR is its simplicity and low computational overhead.

Algorithm 1 Mistral's algorithm, code of node n_i.

```
1: Initialisation:
2:   DpBuffer ← ∅                                  {Received dps}
3:   cp ← ⊥                                        {Compensation packet}
4:   CpBuffer ← ∅                                  {Received cps}

5: upon flood(dp) do
6:   broadcast(dp)

7: upon reception of data packet dp for the first time do
8:   process(dp)
9:   runRecovery(dp)

10: upon reception of compensation packet cp from sender p_j
    do
11:   if cp.ids contains unknown dp ID then
12:     if recovery possible then
13:       dp_recov ← recover from cp
14:       process(dp_recov)
15:       runRecovery(dp_recov)
16:     else
17:       CpBuffer ← CpBuffer ∪ {cp}
18:       if level-2 recovery then
19:         expand(cp)
20:         for all recovered dp do
21:           process(dp)
22:           runRecovery(dp)

23: procedure process(dp)                          {handles a data packet}
24:   DpBuffer ← DpBuffer ∪ {dp}
25:   if F(dp) then
26:     broadcast(dp)
27:   else
28:     composeCompensationPac(dp)
29:   deliver dp to the application

30: procedure composeCompensationPac(dp)           {constructs a
    cp}
31:   cp.payload ← cp.payload ⊗ dp.payload
32:   cp.ids ← cp.ids ∪ {dp.id}; cp.ttls ← cp.ttls ∪ {dp.ttl}
33:   if |cp.ids| ≥ c then{|X| returns the nbr of elements in
      X}
34:     broadcast(cp)
35:     cp ← ⊥

36: procedure runRecovery(dp)                      {recovers dps from
    CpBuffer}
37:   for all cp1 ∈ CpBuffer do
38:     if dp.id ∈ cp1.ids then
39:       remove dp from cp                        {including TTL and ID}
40:       if recovery from cp1 possible then
41:         dp_recov ← recover from cp1
42:   for all recovered data packets dp'_recov do
43:     process(dp'_recov)
44:     runRecovery(dp'_recov)

45: procedure expand(cp)                           {level-2 recovery}
46:   for all cp1 ∈ CpBuffer do
47:     for all cp2 ∈ CpBuffer ∧ cp2 ≠ cp1 do
48:       if cp1 or cp2 is reducible then
49:         cp ← reduction from cp1 and cp2
50:         CpBuffer ← CpBuffer ∪ {cp}
```

If the receiver of a compensation packet already has all but one of the contained data packets, the compensation packet will allow the reconstruction of that missing data packet. However, the recipient of a compensation packet has no a-priori way to know what data packets were used to build the compensation packet. Accordingly, compensation packets must include a list of all its contained data packet IDs. Assuming IP-style node addresses, the sender ID is represented by four bytes. The local sequence number consists of one byte, which allows Mistral to send 255 flooded messages by a node before looping back to 0. From this, we can see that the size of a compensation packets will be the payload size plus five times the number of included data packets c, i.e, $|cp| = |payload_{dp}| + 5 * c$. Notice that the packet size is independent of the number of nodes in the system as a whole. This information is sufficient for floodings that span the entire node field.

A complication arises in applications where the the scope of flooding is limited by a time-to-live (TTL) parameter. Here, the compensation packets need to represent the TTL for each contained data packet; otherwise, if a node recovers data packet dp from a compensation packet, it has no way to know what TTL to use when rebroadcasting dp. If it chooses a TTL that is smaller than the true TTL, then the flooding may die out too early. If the TTL is too high, then valuable bandwidth is wasted. Even worse, if the flooding is a part of a routing mechanisms and the routing mechanism depends on the TTL, then loops occur in the routing paths.

Clearly we cannot treat the TTL of a data packet as a part of that packet's payload, since TTLs are decremented at every hop of the data packet. The problem here is that incoming TTLs for received packets might differ at the node undertaking the reconstruction relative to the node that built the compensation packet. Thus, TTLs need to be added to the compensation packet outside of the payload.

The simplest approach is to add a list of TTLs to the compensation packet. Since the TTL is generally represented by one byte another c bytes are added to the size of a compensation packet. In effect, the TTL extends the packet-id by one byte.

Unfortunatly, this simple approach adds additional overhead, which we would like to avoid. A first point to notice is that TTLs are often defined based on some estimate and are thus, by design, already an approximation. Hence, if we manage to limit the error to some low number, we can manage with an approximate recnstruction of the TTL value. For instance, we could store the sum of all TTLs. The TTL of a recovered data packet can then be restored by subtracting the TTL's of all known packets (all data packets except one). To limit the size to one byte, we apply the modulo operator to this sum. Using this approach, the error will in most cases be within ± 1, or in total $\pm c$, which is acceptable for most applications. Thus, the total size of a compensation packet is $5c + 1 + |payload_{dp}|$ bytes.

Although we have not explored the idea yet, it may be possible to further reduce the overhead associated with compensation packets by compressing packet-id information. For example, in a MANET where most communication originates with a very small set of senders, we could assign those senders some sort of very small id. Moreover, it may sometimes be possible to compress the compensation packet payload itself. On the other hand, such ideas increase the computational overhead at the receiver and hence would require careful evaluation.

3.2.2 *Recovering from Compensation Packets*

To recover data packets from compensation packets we use a two-level recovery mechanism. The first level recovers data packets based on the data packets that have already been received. If $c - 1$ data packets contained in a compensation packet are known, the missing one can be reconstructed. Compensation packets that contain two or more missing data packets are stored (in the $CpBuffer$) and reconsidered when new data packets arrive or are recovered from other compensation packets. Actually, we do not store complete compensation packets, but only compensation packets that contain the IDs, TTL(s), and payload of the missing packets. More specifically, we xor the known data packet payloads with the payload of the compensation packet. After some time compensation packets are garbage collected, as it has become highly unlikely that the missing data packet(s) will be received in the future.

The level-2 recovery mechanism is more elaborate. Instead of only considering incoming and recovered data packets this algorithm also matches compensation packets against each other. The matching operation works as a reduction. Each new compensation packet is compared with all stored compensation packets. If either one of the packets is completely contained in the other, then a new compensation packet is added, which contains the set of data packet IDs of the larger packet minus the ones in the smaller packet. The new payload is constructed by applying XOR to both compensation packets. Provided that it does not allow the immediate recovery of a data packet, this reduced compensation packet is then added to the set of stored compensation packets (in $CpBuffer$).

Clearly, level-2 recovery adds a considerable overhead, both in storage and computation. Its application thus makes sense only if the gain in recovered data packets is significant with respect to level-1 recovery. We explore level-1 and level-2 recovery using simulations in Section 5.2.3.

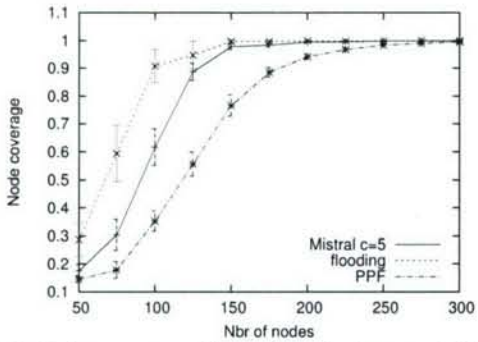
3.2.3 *Mutable Payloads*

Many routing protocols modify the flooded packets during the flooding. We have already shown how to handle TTL values. But some protocols modify other parts of data packets, for example by touching internal parameters, building a route trace, etc. To allow Mistral to handle these cases, we extend the above mechanism into compensation packets that include a mutable part and an immutable part of the payload. Clearly, the larger the immutable part is relative to the mutable part, the better the performance of Mistral. This is particularly the case as the immutable parts can be reduced into an immutable part of the same size, while mutable parts need to be appended to each other, thereby resulting in a size of $\sum_{i=0}^{c} mutablePartOf(dp_i)$. In genernal, the size of a compensation packet will now be $5c + |immutablePayload_{dp}| + \sum_{i=0}^{c} mutablePartOf(dp_i)$.

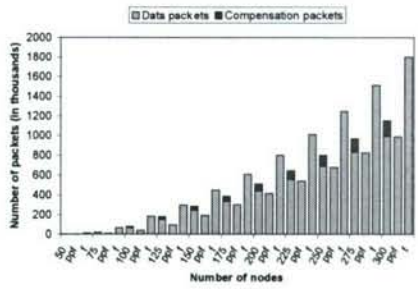
In the evaluation that follows, we assume that packets contain no mutable data other than the TTL.

4. ANALYSIS

In this section, we provide a simple analysis of Mistral. We denote by d_{max} the maximal diameter of the node field and consider floodings that span the entire node field. The maximal transmission latency $t_{maxTrans}$ is the maximal trans-



(a) Node coverage with varying density, $p = 0.55$.



(b) Message overhead with varying density, $p = 0.55$.

Figure 1: Node coverage and message overhead with varying node density.

mission range (88m) divided by the transmission speed. The time needed to do all the computations on a node is Δt, and we assume that there are no delays in the outgoing sending buffers, i.e., that there is no contention in the transmission medium.

Let f_i denote the number of floodings originating at node i, then the estimated overall generated number of compensation packets in a network with n nodes is $G = n\frac{(1-p)\Sigma_i f_i}{c}$, assuming that every node receives all flooded data packet at least once. Thus, the overhead in bytes is $G * (5c + 1 + |payload_{dp}|)$.

Assume that δ_{flood} denotes the average reception frequency of data packet that are received for the first time. Then, the estimated time needed to fill up a c-based compensation packet is $t_{recoveryPac} = \frac{c}{(1-p)*\delta_{flood}}$.

We now consider the delivery latency of a data packet. The worst case occurs when the flooding source and the destination are d_{max} hops apart and the data packet is always forwarded as part of a compensation packet. In this case, the maximum delivery latency is $d_{max} * (t_{recoveryPac} + \Delta t + t_{maxTrans})$, while the estimated maximum delivery latency is $(1-p) * d_{max} * (t_{recoveryPac} + \Delta t + t_{maxTrans}) + p * d_{max} * (\Delta t + t_{maxTrans})$.

We now compute the number of packets sent by a single flooding in a network of N connected nodes. Purely probabilistic flooding has a message overhead of $E(MsgOverhead) = p * N$, if we assume that every node receives the flooded message at least once. Mistral adds an estimate of $\frac{1}{c}$ for every dropped message. Thus, the total overhead per flooding is $(1-p) * N * \frac{1}{c} + p * N$. If the assumption that all nodes receive the flooded message is relaxed then the relative overhead added by Mistral increases. Each node that receives the flooded message only because of Mistral again contributes an additional broadcast or partial compensation to the overhead. Naturally, the additional overhead pays off through the increased node coverage.

5. SIMULATIONS

For our simulation we used JiST/SWANS v1.0.4 [1, 4], a simulation environment for ad hoc networks. Java applications written for a real deployment can be ported to the simulation environment and then placed under a variety of simulated scenarios and loads. JiST/SWANS intercepts the calls to the communication layer and dynamically transforms them into calls to the simulator's communication package.

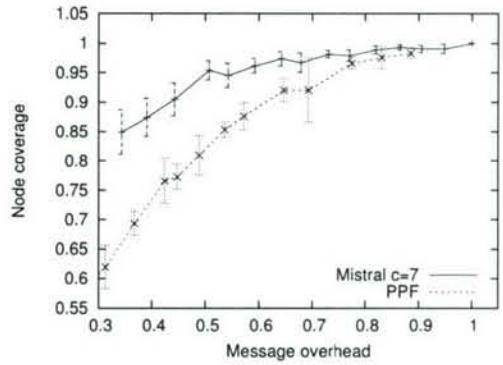
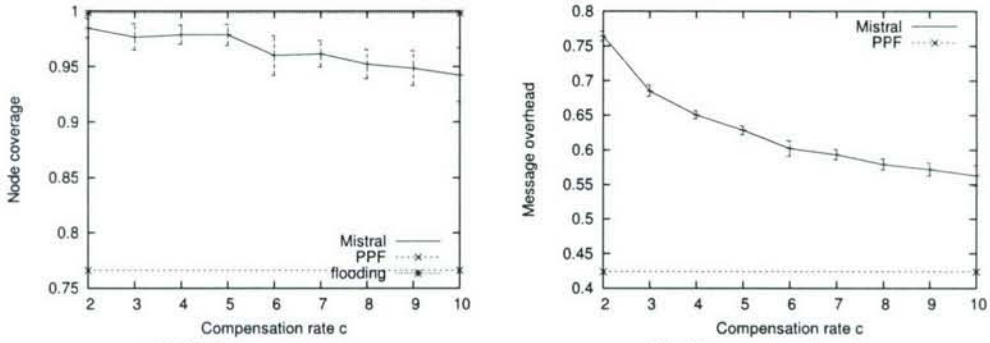


Figure 2: Node coverage with respect to message overhead.

5.1 Setup

We consider a set of nodes. Communication between two nodes m and n occurs in an ad hoc manner and may be asymmetric, i.e., n may be able to communicate with m, but the inverse may not be possible. Communication is by broadcast as defined in the 802.11b standard [12] and can be subject to interference, in which case the message cannot be received. Interference can occur without the sender being able to detect the problem (this is called the *hidden terminal problem* [2]).

We simulate a wireless ad-hoc network with 150 nodes uniformly distributed in a field of size 600x600m. Nodes are stationary, except for one case in which we measure the impact of mobility (Section 5.2.4). The maximal transmission range of a node is set to 88m. Every node starts flooding 20 messages at a regular interval, once all nodes are started up. All flooding occurs across the entire node field. Hence, ideally all nodes should receive all flooded messages.



(a) Node coverage. (b) Message overhead.

Figure 3: Varying compensation rate c, $p = 0.55$.

Our work models disconnections due to mobility, transmission range limits, and the hidden terminal problem just mentioned (using JiST/SWANS' *RadioNoiseIndep* package, which uses a radio model identical to ns2). Unless otherwise mentioned, we use the default values defined in JiST/SWANS.

The nodes start up at random times and positions. When they are all up and running, we start sending the flooding messages and we wait until all messages have been received (terminating simulation).

5.2 Results

In this section, we present the results of our simulation. Every node periodically, every 50s, floods a message throughout the entire field. We have chosen a low flooding rate because in our simulations we want to minimize the effect of packet loss due to buffer overflows and interference. The nodes are added to the sensor field at time 0s but start flooding at times uniformly distributed between 0 and 60s. All results give the average over at least 30 runs in different uniform node distributions. In general, the variance in the simulation results for ad hoc networks is high. This is due to the many sources of randomness: distribution of the sensor nodes, the paths of nodes, the time the nodes flood a message, etc. Thus, where significant we indicate the 95%-confidence intervals (CI).

To evaluate the quality of Mistral, we are mainly interested in two properties: node coverage and message overhead. *Node coverage* measures the number of nodes that have received the messages, while message overhead indicates the total number of sent messages. Both measurements are normalized against a connected network with the same number of nodes. In a connected network, any node can communicate with any other node. Thus, node coverage is given as a percentage of all nodes in the network, while message overhead is given as the percentage of the message overhead in the case in which all nodes receive all messages (normal flooding). Note that the message overhead in the connected network equals the product of the number of flooded messages with the number of nodes. Moreover, it is generally lower in a network with partitions. Since Mistral complements local-knowledge-based approaches and is based on purely probabilistic flooding, we compare Mistral to the latter. Purely probabilistic flooding is entirely defined by the rebroadcasting probability p. For completeness, we also show the results for simple flooding, which corresponds to PPF with $p = 1.0$.

In the following, we evaluate the following properties of Mistral: its behavior in the face of varying density, varying protocol parameters, node mobility, packet loss, and with the secondary recovery mechanism. Unless explicitly stated otherwise, we use the above default values in our measurements.

5.2.1 *Impact of Density*

We start by measuring the impact of node density on the node coverage and the message overhead. Fig. 1(a) shows the node coverage with varying number of nodes. It shows three measurements: simple flooding, purely probabilistic flooding (PPF), and Mistral with compensation rate $c = 5$. The rebroadcast probability is set to $p = 0.55$ in the cases of purely probabilistic flooding and Mistral. As expected, Mistral has a much higher node coverage than purely probabilistic flooding, especially for lower node densities. If the node density passes a certain threshold (around 225 nodes for Mistral), it is sufficiently high such that all nodes receive all messages. In contrast, with low density only a low percentage of the nodes receive all messages. However, below a certain threshold (around 150 nodes) even simple flooding cannot reach all nodes.

In Fig. 1(b) we show the corresponding message overhead. For every number of nodes indicated on the x-axis, we draw the sent number of packets for Mistral, purely probabilistic flooding (ppf), and simple flooding (f). Mistral's packets are further separated into data packets and compensation packets. Since Mistral adds additional compensation packets, its total message overhead is higher than the one of purely probabilistic flooding. Notice also that for low densities the number of flooding packets is higher. Due to higher node coverage in Mistral, more nodes receive the message and thus more nodes also rebroadcast the message, which accounts for the higher number of flooding packets compared to PPF.

Thus, to measure Mistral's net gain in node coverage, as compared to purely probabilistic flooding, we need to consider both node coverage and message overhead graphs. Indeed, since Mistral's compensation mechanism adds an additional overhead, we cannot directly compare the two approaches with the same rebroadcast probability p. Rather,

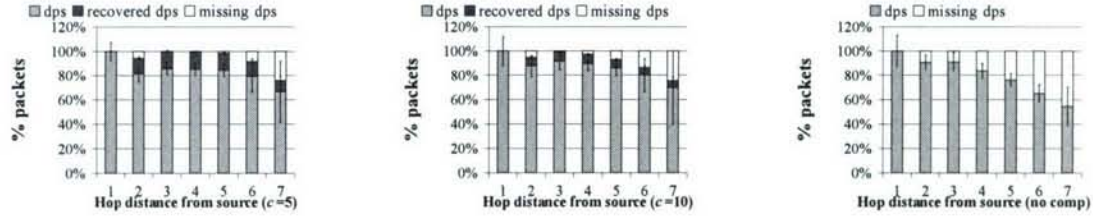


Figure 4: Recovery based on hop counts, single source and $p = 0.55$.

we need to compare Mistral with the purely probabilistic flooding using a rebroadcasting probability with a similar message overhead. Fig. 2 plots the node coverage with respect to the message overhead, for $c = 7$. The message overhead corresponds to simulation runs with p varying from 0.3 (0.4 for PPF) to 1, in steps of 0.05. The gain with Mistral is especially prominent for low rebroadcast probabilities p. Of course, low rebroadcast probabilities lead to many dropped rebroadcasts and thus the node coverage becomes low. Using Mistral allows some of the nodes to recover messages they may have missed. For an overhead of 0.35, Mistral improves the node coverage by 20%, for an overhead of 0.55 by 10%, and for overhead around 0.75 it is closer to 3%.

5.2.2 *Compensation Rate*

We now turn to one of the parameters that determine the behavior of Mistral: compensation rate c. In Fig. 3(a), we show the node coverage with compensation rate c varying from 2 to 10, for 150 nodes and rebroadcasting probability $p = 0.55$. Generally, the node coverage decreases with increasing compensation rate. For comparison, the graph also indicates the node coverage for flooding and purely probabilitstic flooding (PPF) with the same parameters. Both flooding and probabilistic flooding are independent from the compensation rate and thus are represented by a horizontal line. Fig. 3(b) gives the corresponding message overhead. Here, the message overhead decreases with increasing compensation rate. Thus, given a particular node coverage the higher the compensation rate the better. However, a higher compensation rate also increases the message delivery latency. Indeed, data packets that are part of a compensation packet spend more time waiting until the compensation packet is filled with sufficient data packets and may thus be delayed.

5.2.3 *Recovery Performance and Overhead*

Next, we measure the number of recovered data packets with respect to the hop count (see Fig.4). In this simulation, a single node at position [300, 300] periodically floods 1000 messages. We give the results for $c = 5$, $c = 10$, and the case with no compensation (no comp). Since the overall number of received data packets is different depending on the hop-distance of a node to the flooding source, we give the percentage of recovered data packets to all flooding packets that should have been received by the nodes at this hop distance from the source. The percentage of recovered data packets is approximately the same for most hop distances. An exception is at hop count 1, where all nodes generally receive the flooded message, because the source floods the data packet with $p = 1.0$ and at a time of low traffic. As fewer compensation packets are sent in the case of $c = 10$, the percentage of recovered data packets is lower compared to the case of $c = 5$. Towards very high numbers of hop counts, no compensation packets are received. However, these nodes are particular cases resulting from a unusual node distribution, which does not occur frequently.

Notice that the percentage of dps increases between $c = 5$ and $c = 10$. The reason is that with smaller c, more compensation packets are sent and the likelihood that a dps is received via a compensation packet increases. Since we count the data packets when they are received for the first time, more packets are received via a compensation packet. Thus, the percentage of dps is smaller for a smaller c.

We use the same setup to measure the packet delivery latency. In contrast to the other simulations, we use a single data point (one random uniform distribution) in this case. The single source floods a data packet every second. The graphs in Fig. 5(a) and (b) show the latency distribution of data packets for $c = 2$ and $c = 8$ with respect to the hop distance of the node. The delivery latency of a data packet is high if it is received by a node only as part of a compensation packet. The higher the compensation rate, the higher this delay is.

Another important characteristic of Mistral is the ratio of compensation packets that cannot be recovered. We say that a *compensation packet is recovered* if all contained data packets have been received or have been recovered. In general, we expect the number of unrecovered compensation packets to increase with increasing compensation rate. The graph in Fig. 5(c) confirms this. It uses our default setup with many flooding sources and shows the total number of received compensation packets and the number of compensation packets that have not been recovered (logarithmic scale on y-axis). Clearly, the lower the compensation rate, the higher the number of sent and thus received compensation packets. This number also includes all compensation packets whose contained data packets have already been received earlier by the receiving node (useless cps). Immediately recovered compensation packets denote the compensation packets that only contain a single unknown data packet. Any compensation packet that contains more unknown data packets is added to *CpBuffer*.

Fig. 5(d) shows the number of sent compensation packets based on the number of nodes in the field. As expected, this

Scalable Technology for a New Generation of Collaborative Applications

MURI Grant Final Report
AFOSR F49620-02-1-0233
May 2002 – April 2007

1. Principal Investigators

Ken Birman (Cornell University)
Al Demers (Cornell University)
Johannes Gehrke (Cornell University)
Keith Marzullo (University of California at San Diego)
Geoffrey M. Voelker (University of California at San Diego)

2. Research Objectives

Our MURI effort emerged from dialog between the AFRL team developing software for the Joint Battlespace Infosphere (JBI) and university researchers at Cornell and elsewhere. It became clear that to be successful, the JBI needed to break completely new ground in offering publish-subscribe capabilities on a scale never previously attempted, and do so with guarantees of security, reliability and predictable performance of a sort impossible for existing commercial products. The AFRL JBI team reacted to this unique challenge by evaluating a number of candidate "core" technologies for distributed systems that map in clear ways to the technical needs of the JBI:

- Group communication (multicast) systems and commercial publish-subscribe systems have a direct correspondence to the communications needs of the JBI.
- The JBI repository will be a database that can be updated and queried in real-time, and there are thus parallels between repository functionality and commercial products in the database area.
- Because many JBI information sources provide data streams or periodic data bursts (indeed, few are likely to be static), there is a strong connection between JBI reporting functionality and the technology of distributed data mining and triggered actions.
- The JBI will use off-the-shelf Web Services technologies wherever possible, thus the degree of match between the JBI and such systems must also be better understood.

It quickly became clear that no existing commercial product is adequate for the full spectrum of JBI requirements. While a number of prototypes have been constructed for various components of the JBI using readily available technologies, these integrate poorly and lack the distributed computing functionality and scalability properties required of the real system. On the basis of commercial experience, any JBI system built in this manner will be fragile and easily disrupted under stress, and omit

important functionality (such as integration of the publish-subscribe aspects of the JBI and the repository) that are lacking in existing commercial offerings. Accordingly, AFRL encouraged the research community to look both at the technical needs of the JBI and its sibling systems in the Navy, Army and Marines, and also to work towards the elaboration of a scientific basis for reasoning about systems such as the JBI - a science some are dubbing "Infospherics" because of its applicability to the Infosphere.

Particularly important are demonstrations of scalable solutions which can be counted upon to remain stable under stress and to offer predictable performance and reliability even when the system is disrupted by an adversary while performing mission-critical tasks. These needs extend from the publish-subscribe functionality per-se to other aspects, such as querying data residing in networks of sensors. As the JBI is adapted for use by other services, some of these aspects may require urgent attention. For example, the Navy is considering the JBI as a platform for future USW sensor applications, but these are primarily data mining problems, not publish-subscribe applications. Understanding how to make such solutions coexist in a single platform is vital for the JBI to emerge into the desired role for the Air Force and its sibling military services.

At Cornell and the University of San Diego, our team came together to pursue common interests at the intersection of large-scale distributed computing systems, Web Services and similar emerging architectures, data mining, and distributed database systems. Our group spans the technology areas needed by the JBI and is united by a shared interest in similar kinds of technologies (particularly probabilistic approaches with good scaling properties), similar application models (relating to publish-subscribe, data mining, and other forms of communication patterns best described as forms of query evaluation), and extensive expertise in experimental evaluation of the technologies we develop.

Birman and van Renesse jointly head the Spinglass group, which is a broad effort developing a new family of reliable, scalable protocols for communication, monitoring, management and control in large distributed systems. Spinglass exploits "multipeer" communication to achieve scalable, probabilistically reliable solutions to component problems that arise in a variety of setting. These components can then be used as tools for enabling direct, live collaboration between participants who may be spread across a global network and using platforms with differing communications capabilities.

Birman and his group focused on scalable group communication systems that provide critical properties such as data replication, distributed coordination, and automated reaction to faults. They developed systems that provide these properties and scale publish-subscribe and group multicast services in many dimensions. They also developed tools for transforming standard Web services into ones that scales to clusters in data centers, automatically replicates data to improve peformance and reliability, and efficiently updates replicated data. Van Renesse and his group targeted the problem of hardening group communication systems to failures and attacks. They developed reliable techniques for detecting intruders in group communication systems and mitigating the damage that they can cause, such as providing incorrect information or dropping packets.

Gehrke and Demers investigated scalability and information management issues in the JBI and other military information dissemination systems. They worked on the challenges of building very large scale

databases which are spread among many sites and yet maintain strong forms of consistency and can be queried in a manner similar to the ways that centralized databases can be queried. They targeted emerging architectures where actual databases reside at the leaves, are updated using transactions, and can support triggered upcalls when events change in ways of interest. Within this overall effort, Gehke's group has been focused on extending existing database systems and building completely new ones optimized for use in distributed sensor networks and other systems that generate high volumes of data rapidly. Demers focused on understanding the behavior of systems built using probabilistic protocols and the risks associated with using approximate algorithms, developing a number of such algorithms for solving simple problems like solving distributed range queries over complex sensor networks, computing aggregate values in large systems, and optimal resource location in distributed settings.

Marzullo and Voelker in the UCSD group focused on providing high service availability and efficient reliability for large-scale distributed systems that form the foundation of JBI efforts. Voelker and his group addressed the problem of providing highly-available services in distributed systems composed of relatively unavailable components. A fundamental challenge for designing and building next-generation distributed systems is providing high availability using these highly unavailable components. Their work developed middleware that provides an abstraction of a highly-available platform to upper-layer software services while running on intermittently available components. They demonstrated this approach in a prototype wide-area file system. In terms of reliability, Marzullo and his group pursued the development of and insights into the theoretical understanding of distributed systems with dependent failures. Their work has developed new solutions to several well-known problems in distributed computing that are optimal when failures are not independent and do not have identical distributions. Making their theoretical results practical, they applied dependent failure system models to a system called Phoenix that cooperatively protects data from loss arising from Internet catastrophes from propagating malware such as viruses and worms.

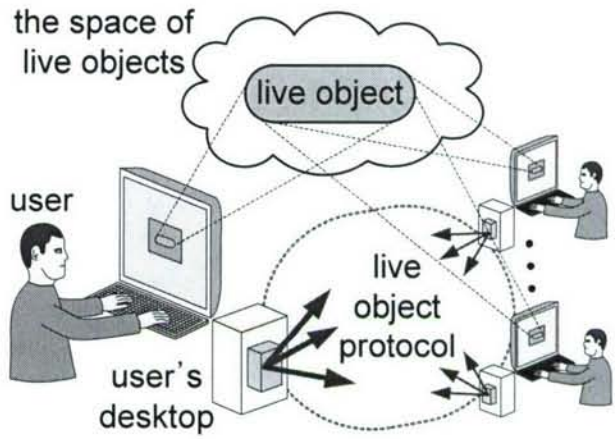
The JBI has proved to be an outstanding focus for our collaboration, and we embarked on a research agenda to use the MURI funding, together with other funding available to us directly from the JBI community and from other sources, such as DARPA, to assist AFRL in taking a major step forward on these challenging and extremely important questions. Over the period of the MURI grant, we established substantial groundwork for a scientific treatment of the military's pressing challenges, backed by rigorous experimental work that helps clarify the uncertainty concerning just how a JBI system might actually be implemented. We believe that the research results from this MURI effort will thus pave the way for future commercial efforts to provide concrete deliverables to the JBI and similar military projects in other services.

3. Status of Efforts, Accomplishments, and New Findings

To better encapsulate the major research accomplishments performed by our group, we next provide individual summaries of the goals, approaches, accomplishments, and impacts on military needs of our efforts funded through this MURI effort.

QUICKSILVER SCALABLE MULTICAST

Goal: *GIG/NCES platforms provide excellent support for point-to-point communication in a web services paradigm, but support for scalable group communication, data replication, distributed coordination and automated reaction to faults are sorely lacking. Our goal in the Quicksilver project is to overcome the inherent scalability problems that limited development of solutions having these properties. The basic premise is that if we can show how to scale a publish-subscribe or group multicast system in many dimensions, in a web services framework, we can then take the next step and build the missing tools.*



Approach: *We developed Quicksilver Scalable Multicast, and found a new way to embed the technology into the Windows and Linux platforms based on what we call a "live objects" interface. The key ideas were as follows:*

- *A live object extends the normal Windows support for component integration to permit a new kind of component in which members belong to a group that replicates data using high-speed multicast. Various back-end communication "drivers" can provide the multicast; QSM is just the first of these*
- *We constructed an implementation of such a multicast protocol, QSM, and have demonstrated that it can scale far beyond the limits of any prior multicast technology, particularly when large numbers of users employ the system and this results in a pattern of extensively overlapped multicast groups. QSM maintains extremely high performance even under disruptions and other forms of injected stress.*
- *We've begun to extend QSM with a new high-level language so that reliability can be layered over the core system in a simple, easily used manner that can support everything from best-effort reliability to strong models such as virtual synchrony or transactional one-copy serializability.*

Accomplishments: *Our work on QSM is attracting attention from major vendors such as Cisco and Microsoft, even as we develop a small user community of early adopters. With the expected release of our live objects layer in the fall of 2007, we should see a burst of users drawn to our system by the ability to create live documents and applications by dragging and dropping live objects onto web pages, into databases, or in Word documents. The basic idea is that by creating such a document and then sharing it, non-programmers can create sophisticated distributed applications much as they build PowerPoint presentations.*

We are also finding that the Web Services standards community is interested in our approach. A journal article proposing a way to extend WS-NOTIFICATION and WS-EVENTING to offer greater flexibility will appear in the fall.

Our work on Quicksilver will continue beyond the termination of the MURI effort under funding from AFRL, AFOSR and other sources.

Impact on the warfighter: Quicksilver enables a new kind of agile response to rapidly evolving conditions. Today, it would be impossible to imagine building, say, a customized application for a search-and-rescue mission on the fly, in the field. With live objects and Quicksilver it may be possible for tomorrow's commander to create custom information solutions as needed, in real-time, for instant deployment to the troops for whom accurate timely information can determine mission success.

References: (For downloads and complete list, visit http://www.cs.cornell.edu/projects/quicksilver/)

Scalable Publish-Subscribe in a Managed Framework. Krzysztof Ostrowski, Ken Birman. Cornell Technical Report (TR2007-2086). October, 2006.

The QuickSilver Properties Framework. Krzysztof Ostrowski, Ken Birman, Danny Dolev. OSDI'06 poster session, Seattle, WA, November 2006.

Properties Framework and Typed Endpoints for Scalable Group Communication. Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report TR2006-2062 (July, 2006).

Scalable Group Communication System for Scalable Trust. Krzysztof Ostrowski, Ken Birman. In Proceedings of The First ACM Workshop on Scalable Trusted Computing (ACM STC 2006). Fairfax, VA. November 3, 2006.

Extensible Web Services Architecture for Notification in Large-Scale Systems. Krzysztof Ostrowski and Ken Birman. In Proceedings of the 2006 IEEE International Conference on Web Services (ICWS 2006). Chicago, IL, September 2006

Declarative Reliable Multi-Party Protocols Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Implementing High-Performance Multicast in a Managed Environment Krzysztof Ostrowski, Ken Birman, Danny Dolev. Cornell University Technical Report (TR2007-2088). April, 2007.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems. Ken Birman, Anne-Marie Kermarrec, Krzysztof Ostrowski, Marin Bertier, Danny Dolev, Robbert Van Renesse. Distributed Event Processing Systems and Architecture Workshop (DEPSA). June 2007.

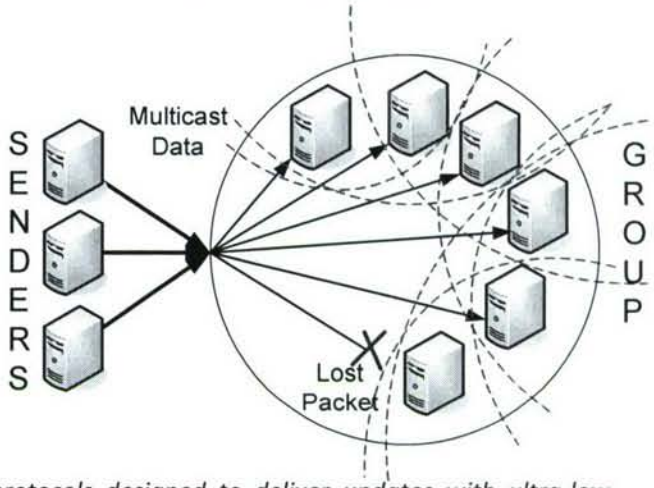
Live Distributed Objects: Enabling the Active Web Krzysztof Ostrowski, Ken Birman, Danny Dolev. To Appear in IEEE Internet Computing.

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. To Appear in the International Journal of Web Services Research. Volume 4, Number 4. October-December 2007.

RICOCHET SCALABLE TIME-CRITICAL MULTICAST PROTOCOL

Goal: *Data center systems are challenged by the difficulty of rapidly disseminating time-critical information, for example within a service running on multiple nodes in a cluster and where the data will trigger some real-time response. Such problems arise in many settings, including radar tracking systems, weapons targeting, real-time control of autonomous vehicles, etc. Moreover, there are many settings in which "standard" web services need to offer rapid end-user response time, even as updates flow into the core system.*



Approach: *Ricochet/Plato are a new family of multicast protocols designed to deliver updates with ultra-low latencies in clusters or data centers hosting scaled servers, again under the assumption that the services are implemented using web services standards. The key idea here is to aggregate traffic so that error correction codes can be computed more quickly than if each data stream was treated separately. Ricochet and Plato explore this in a multicast setting; we are currently taking the next step by developing, Maelstrom, which applies similar ideas with an emphasis on connections between data centers over long-distance WAN links. Cisco and Microsoft have shown keen interest in using these solutions in their respective platforms and products, and Raytheon is helping us explore transition into military platforms using the DDS standard.*

Accomplishments:

- *Designed and implemented the Ricochet protocol, undertook a comprehensive evaluation, and completed a series of papers on this work, including mobile wireless applications (Mistral).*
- *Through dialog with companies in the web services community, identified promising technology transition opportunities. Raytheon is taking the lead on pursuing these with us, focusing on the military-standard DDS communication architecture.*
- *Made our solutions available to other vendors, including the Apache web services platform.*
- *Developed and implemented a predictive real-time ordering protocol, Plato.*

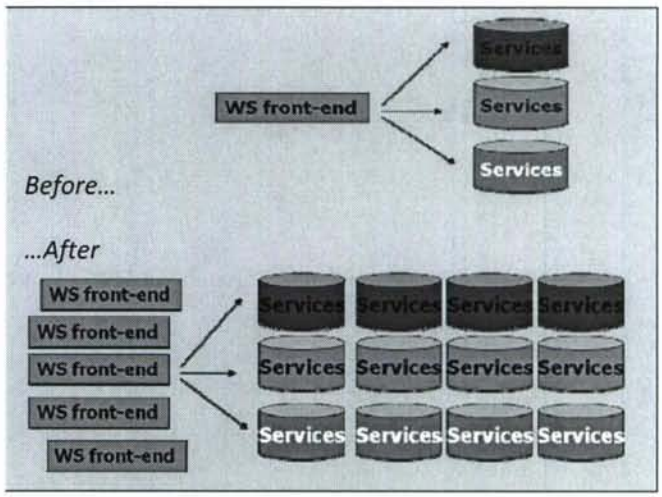
Impact on the warfighter: During military operations, timely information can have life or death implications. Ricochet makes possible dramatically improved responsiveness and maintains this guarantee as it scales up.

References: *Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*

Ricochet: Lateral Error Correction for Time-Critical Multicast. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. To Appear in Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). Cambridge, MA. April 2007.
Mistral: Efficient Flooding in Mobile Ad-hoc Networks. S. Pleisch, M. Balakrishnan, K. Birman, and R. van Renesse. In Proceedings of the Seventh ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM MobiHoc 2006). Florence, Italy May 2006.
PLATO:Predictive Latency-Aware Total Ordering. Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. In Proceedings of the SRDS 2006: 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK. October 2006.

TEMPEST: A TOOL FOR CREATING SCALABLE WEB SERVICES

Goal: *Today it is much too difficult for programmers with a typical MEng-level of training to implement scalable, self-managing web services that can run on datacenters in GIG/NCES settings and that will automatically adapt as conditions change. By solving this problem we can reduce the delays and costs associated with implementing new services for the GIG while also ensuring that those services will be seamlessly adaptive even under stress. Moreover, we can bring best-of-breed solutions to the table in a reusable form, reducing the tendency of vendors to produce stovepipe technologies that only the developer can extend or support.*



Approach: Tempest is a tool for turning a fairly vanilla web service into one that scales on a cluster or data center, has automatically replicated data, and employs Ricochet to send updates. Gossip communication is employed as an adjunct to this to repair any inconsistency that might arise as a result of a failure. Tempest is just reaching a stage at which early demos are feasible; the system is able to take a front end (for example an application that builds web pages) and a set of back end web services and will automatically replicate each of these, to varying degrees, in a manner that achieves predictable time-critical response even under stress, even when faults occur, and even when the services have very different behavioral profiles (such as mean response time, etc).

Accomplishments: Tempest was completed in early 2006 and works well; it uses a novel gossip-based approach to propagate updates, detect and repair inconsistencies, and for self-management of the clustered web application. In current work, we are exploring an extension in which Tempest could be used to create integrated enterprise web applications, still in a highly automated manner, with our gossip protocols used to extract data from source applications, replicate it across a WAN, and then integrate it with an application needing to import that data remotely.

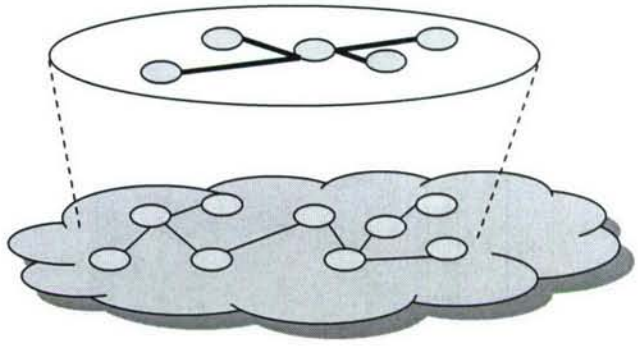
References: *Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*

Scalable Multicast Platforms for a New Generation of Robust Distributed Applications. Ken Birman, Mahesh Balakrishnan, Danny Dolev, Tudor Marian, Krzysztof Ostrowski, Amar Phanishayee. To Appear in Proceedings of the Second IEEE/Create-Net/ICST International Conference on Communication System software and Middleware (COMSWARE). Bangalore, India. January 7-12, 2007.

A Scalable Services Architecture. Tudor Marian, Ken Birman, and Robbert van Renesse. To appear in Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2006). Leeds, UK. October 2006.

Fireflies: Scalable Byzantine Overlay Networking

Goal: *"Fireflies" is a scalable protocol for supporting intrusion-tolerant network overlays. While such a protocol cannot distinguish Byzantine nodes from correct nodes in general, Fireflies provides correct nodes with a reasonably current view of which nodes are live, as well as a pseudo-random mesh for communication. The amount of data sent by correct nodes grows linearly with the aggregate rate of failures and recoveries, even if provoked by Byzantine nodes. The set of correct nodes form a connected sub-mesh, and Byzantine nodes cannot eclipse correct nodes.*



Approach: *Providing each member with membership is a form of agreement. Previous works on Byzantine fault-tolerant agreement establish invariants that are impossible to invalidate. Even the most practical of these protocols require several rounds of all members broadcasting state to all other members, and can consequently not scale up to more than perhaps a few dozen members. In order to scale to thousands or more members, we had to come up with a radically different approach. Fireflies makes use of epidemic techniques ("gossip") to form a probabilistic agreement, which can only establish invariants that hold with a certain probability. Because invariants never hold for certain, defense against adversaries trying to break agreement can never rest.*

Accomplishments: *Fireflies has been adopted by several research projects around the world. For example, at UT Austin Prof. Alvisi and Dahlin are working to create scalable Byzantine and Rational fault-tolerant communication systems making use of game theory (incentives). While their work has been successful, they were not able to deal this far with dynamic systems in which members could come and go. They are now building on Fireflies to remedy this shortcoming of their work. In Norway, Prof. Johansen has developed a system, based on Fireflies, to dispatch security updates in a timely and coordinated manner. Traditional software updates are vulnerable to reverse engineering to discover software flaws. In Israel, Prof. Dolev is developing a self-stabilizing version of Fireflies in order to add additional immunity to attacks. At Cornell itself, Fireflies forms the basis for the SecureStream video streaming system (reported separately).*

Impact on the warfighter: Modern warfighter equipment will almost certainly carry devices that employ state-of-the-art distributed communication protocols. If one or more such devices were compromised, most such protocols could be easily attacked without the warfighter being able to tell the difference. As a result, a warfighter cannot put much trust in these devices. Fireflies is much less susceptible to such attacks and as a result allows warfighters to put significantly more trust in devices that use Fireflies.

References: (For downloading Fireflies, visit http://sourceforge.net/projects/fireflies)

Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays. Håvard Johansen, Andre Allavena, and Robbert van Renesse. Eurosys 2006. Leuven, Belgium. April 2006.

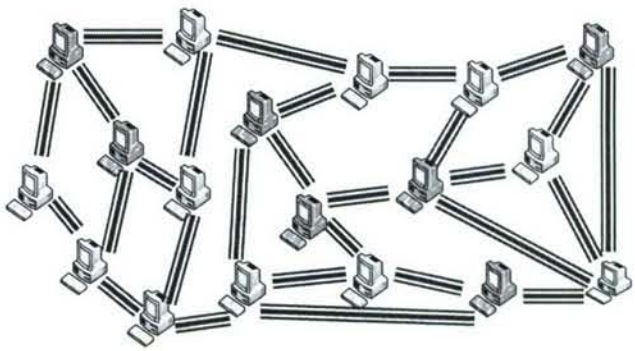
Correctness of Fireflies. Andre Allavena and Robbert van Renesse. Internal Report. June 2006.

FirePatch: Secure and Time-Critical Dissemination of Software Patches. Håvard Johansen, Dag Johansen, and Robbert van Renesse. IFIP International Information Security Conference (IFIPSEC 2007). Sandton, South-Africa. May 2007.

Self-Stabilizing and Byzantine-Tolerant Overlay Network. Danny Dolev and Robbert van Renesse. Submitted to OPODIS 2007. July 2007.

SecureStream: Intrusion-Tolerant Video Streaming

Goal: *Application-level multicast systems are vulnerable to attack that impede nodes from receiving desired data. Live streaming protocols are especially susceptible to packet loss induced by malicious behavior. We describe SecureStream, an application-level live streaming system built using a pull-based architecture that results in improved tolerance of malicious behavior. SecureStream is implemented as a layer running over Fireflies, an intrusion-tolerant membership protocol.*



Approach: *Our work introduces several techniques that reduce the opportunity for an attacker to compromise the quality of a streaming session, without incurring a high computational or network overhead. To repel forgery attacks, we employ an efficient packet authentication technique based on computing and distributing verification digests. To prevent attacks on the overlay structure (the membership protocol on top of which multicast systems operate), SecureStream is built upon Fireflies, a scalable one-hop Byzantine membership protocol.*

Accomplishments:

- *SecureStream is the first exploration of end-system attacks in the context of live streaming peer-to-peer protocols.*
- *We leverage previous work and present a comparison of different authentication protocols for signing and verifying packets efficiently in the context of application-level multicast.*
- *We thoroughly evaluate the effects of internal malicious peers on pull-based protocols. The issue is more serious than has previously been recognized.*
- *Aspects of our work are being adopted by other research groups. For example, at UT Austin Profs. Alvisi and Dahlin are using our linear digests. Prof. Porto at UFRGS in Brazil is extending SecureStream to deal with heterogeneity in the system.*

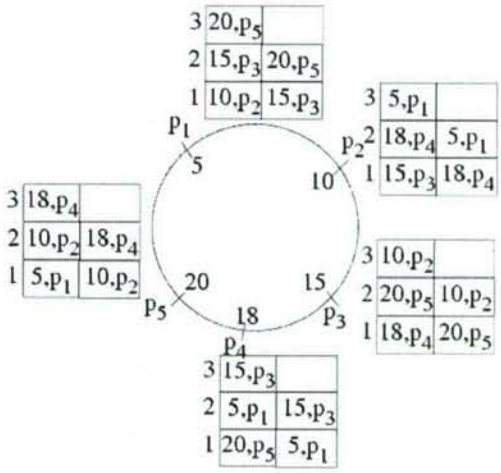
Impact on the warfighter: SecureStream can provide reliable delivery of streaming media to the warfighter even in the face of cyber-attacks.

References: (*Our software download site is http://www.cs.cornell.edu/projects/quicksilver/*)

Maya Haridasan and Robbert van Renesse. *Defense Against Intrusion in a Live Streaming Multicast System*. 6th IEEE International Conference on Peer-to-Peer Computing (P2P2006). Cambridge, UK. September 2006.
Maya Haridasan and Robbert van Renesse. *SecureStream: An Intrusion-Tolerant Protocol for Live-Streaming Dissemination*. The Journal of Computer Communication's Special issue on Foundation of Peer-to-Peer Computing (accepted for publication). 2007.

P-RING: SCALABLE DISTRIBUTED RANGE QUERIES

Goal: *Modern data management systems are increasingly challenged by the need to support very large scale data sets distributed among many sites. The systems must allow distributed data to be queried in a manner similar to the ways centralized databases are queried. They must perform rapid dissemination of time-critical updates while providing strong consistency guarantees for concurrent queries, so that incoming data can reliably trigger a real-time response. Finally, the systems must be fault-tolerant, able to withstand high rates of churn with minimal effect on query performance or correctness.*



Approach:

We have developed P-Ring, a new peer-to-peer index structure that efficiently and scalably supports range queries as well as equality queries, and is robust even under high rates of churn. P-Ring can be viewed as an alternative approach to our earlier Kelips work. The Kelips design tolerates somewhat increased memory usage – O(sqrt(n)) – in exchange for O(1) file lookup times. This memory requirement is acceptable for current systems, but could eventually become a scalability bottleneck. P-Ring takes a different approach, using a hierarchy of fault-tolerant rings to provide O(log(n)) lookup time, by a protocol similar to a skiplist, with an improved memory requirement that is only polylog in n. The P-Ring also achieves excellent load balancing. A straightforward scheme yields a worst-case imbalance factor of 2, analogous to a B+ tree; while a more complicated but quite practical scheme can yield arbitrarily small imbalance factors with constant amortized overhead. A prototype implementation of P-Ring outperforms existing systems that attempt to provide similar guarantees.

Accomplishments:

Designed and implemented the P-Ring protocols and performed a thorough experimental evaluation.

- *Published the results in SIGMOD.*
- *Made our prototype implementation available as open source.*
- *Identified promising technology transition opportunities in collaboration with a local company (ATC-NY). Ongoing work is extending the system to a native XML database.*

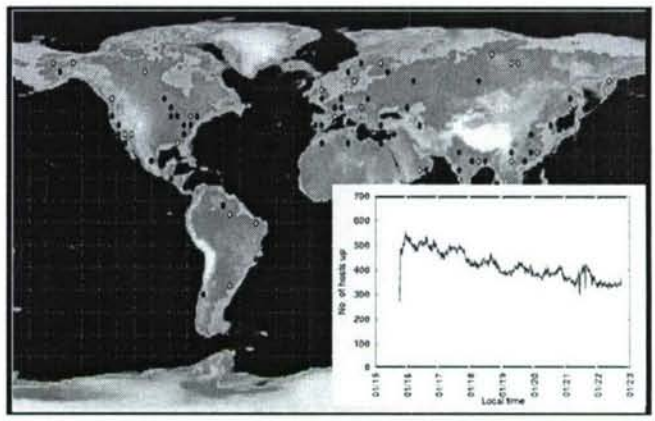
Impact on the warfighter: During military operations, timely and correct information can be vitally important. This work provides scalable correctness and responsiveness guarantees even under high rates of churn.

References:

P-Ring: An Efficient and Robust P2P Range Index Structure. Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke and Jayavel Shanmugasundaram. Proceedings 2007 ACM SIGMOD Conference. Beijing, China, June 2007.

Automated Availability Management

Goal: *A number of issues arise as Web Services and similar COTS components are migrated into GIG/NCES platforms. Our goal for developing automated availability management is to address the problem of providing highly-available services in distributed systems composed of relatively unavailable components. This problem arises in many settings, including ad-hoc wireless networks where disconnection and reconnection is frequent (e.g., battlefield scenarios), sensor networks (e.g., intelligence gathering), distributed computation and storage, etc.*



Approach: *A distinguishing feature across these disparate types of distributed systems is that they are composed of relatively unavailable components. Rather than being always available until failure, the components in these systems are both available and unavailable on a frequent basis (daily, even hourly), yet remain a functioning component of the system on long-term time scales (weeks to months to years). A fundamental challenge for designing and building next-generation distributed systems is providing high availability using these highly unavailable components. Automated availability management formalizes availability as an explicit property in distributed systems. Users and applications can request explicit availability guarantees for system objects and resources. For instance, in a wide-area distributed file system, users would specify that by default files require 99.9% availability over two years. To provide such guarantees, automated availability management uses (1) availability models to make efficient resource provisioning decisions and to predict and estimate future resource availability; (2) redundancy mechanisms to mask and tolerate component unavailability; and (3) repair policies to dynamically maintain resource availability in response to intermittent and permanent failure.*

Accomplishments: *We developed and evaluated a range of availability models, redundancy mechanisms, and repair policies across a spectrum of system configurations. As a concrete point in the system environment space, we measured the temporary and permanent failure characteristics of the Overnet peer-to-peer file sharing overlay network. This work was the first to study these characteristics in such systems, and the traces we gathered were used by many other research groups to evaluate their own efforts. We also refined automated availability management to exploit resource to improve system performance and reliability. Our goal has been to expose resource heterogeneity among nodes and tailor systems to take advantage of it.*

Impact on the warfighter: *Automated availability management provides a convenient abstraction for implementing highly available distributed services in situations where network disconnection and reconnection is frequent. The situations occur at many levels in military deployments, ranging from ad-hoc networks among troop patrols, battlefield communication networks linking troops, vehicles, and air support, carrier groups at sea, and the world-wide GIG IT infrastructure.*

References: (For downloads and complete list, visit http://sysnet.ucsd.edu/projects/recall/)

Replication Strategies for Highly Available Peer-to-Peer Storage Systems. Ranjita Bhagwan, David Moore, Stefan Savage, and Geoffrey M. Voelker. UCSD Technical Report No. CS2002-0726, November 2002.
Understanding Availability. Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Proceedings of the International Workshop on Peer To Peer Systems (IPTPS'03), Berkeley, CA, February, 2003.
Ranjita Bhagwan, Priya Mahadevan, George Varghese, and Geoffrey M. Voelker. Cone: A Distributed Heap-Based Approach to Resource Selection UCSD Technical Report CS2004-0784.
On Object Maintenance in Peer-to-Peer Systems Kiran Tati and Geoffrey M. Voelker. In Proceedings of the International Workshop on Peer-To-Peer Systems (IPTPS), Santa Barbara, California, February 2006.
Maximizing Data Locality in Distributed Systems, Fan Chung, Ron Graham, Ranjita Bhagwan, Geoffrey M. Voelker, and Stefan Savage, Journal of Computer and System Sciences 72(8), December 2006.

TotalRecall File System

Goal: *Our goal was to develop a specific system application of the automated availability approach. We designed a storage system called TotalRecall that applies automated availability management to large-scale, wide-area distributed storage systems. TotalRecall guarantees user-specified levels of data availability while minimizing the overhead needed to provide these guarantees in highly dynamic environments.*

Approach: *TotalRecall predicts the availability of its components over time, determines the appropriate level of redundancy to tolerate transient outages, and automatically initiates repair actions to meet user requirements. It uses replication and erasure coding to adapt the degree of redundancy and frequency of repair to the distribution of failures observed and predicted in the system. It also uses two repair strategies, eager repair and lazy repair, for trading off availability and replication overhead. Moreover, it closely approximates key system parameters, such as the appropriate level of redundancy, from a combination of underlying measurements and requirements. Finally, we have implemented and evaluated a prototype of TotalRecall that automatically adapts to changes in the underlying host population while effectively managing file availability and efficiently using bandwidth and storage resources.*

Accomplishments:

- *Developed and evaluated specific availability management mechanisms for storage systems, including availability models for short-term temporary and long-term permanent failures, erasure coding and mirroring redundancy mechanisms, and eager and lazy repair policies.*
- *Designed and implemented a prototype that runs on the PlanetLab testbed as the TotalRecall File System. Each participating host exports an NFSv3 file system interface to the system. External client hosts can mount TRFS and use it as any other remote NFS file system, storing data with high availability.*
- *Developed ``ShortCuts'', a routing approach for lookups that uses soft state to achieve routing performance that approaches the aggressive performance of one-hop schemes, yet uses an order of magnitude less communication overhead on average.*

Impact on the warfighter: *The TotalRecall File System provides a highly available distributed storage service in situations where network disconnection and reconnection is frequent. It is particularly useful in large-scale GIG IT infrastructures that face challenging communication constraints, such as among the many ships that comprise carrier groups operating at sea, as well as world-wide information infrastructure, such as the storage services supporting the data collection, analysis, and reporting performed by military branches and government agencies.*

References: (For downloads and complete list, visit http://sysnet.ucsd.edu/projects/recall/)

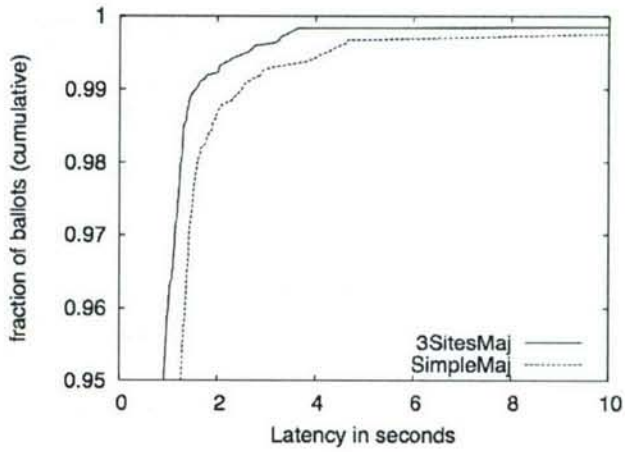
TotalRecall: System Support for Automated Availability Management Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. In Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA, March 2004.

ShortCuts: Using Soft State To Improve DHT Routing. Kiran Tati and Geoffrey M. Voelker. In Proceedings of the Ninth International Workshop on Web Content Caching and Distribution (WCW'04), Beijing, China, October 2004.

Resource Reclamation in Distributed Hash Tables. Kiran Tati and Geoffrey M. Voelker. University of California, San Diego, CSE Technical Report CS2006-0863, July 2006.

Realistic Abstract Failure Models

Goal: *Failure models are part of the contract used in designing and deploying a distributed system. A failure model says what can go wrong with the environment, and so the system needs to be able to sustain its mission in the face of these adverse conditions. From a protocol design point of view, though, a failure model should be simple and abstract, since efficiency is obtained by leveraging off the details of the failure model. Practical details that are often ignored include non-identical failure rates, non-zero covariance of failures, and communications failures arising from BGP convergence issues.*



Approach: *One of the most fundamental protocols for fault tolerance is consensus, and so we deconstructed the various versions of this protocol to understand how it used the simple failure models for which it was developed. We identified two kinds of properties -* core *properties, useful describing failure scenarios, and* survivor set *properties, useful for showing that information is preserved despite failures. The two kinds of properties are duals of each other, and can be generalized to accommodate non-identical failures, non-zero covariance of failures, as well as many other failure patterns.*

Accomplishments: *We have generalized much of the lower bound results for consensus, quorum update, voting, and related problems. The results have been surprising: we have essentially developed a methodology for taking advantage of dependent failures. By knowing which failure patterns are more likely and which are not likely, replication of information can be done in an informed manner. In addition, we have found that some previous lower bounds are serendipitous: some difficulties with more general failure models appear only when dependent failures can occur. In addition, our work has shown that significant performance gains can be obtained using more accurate failure models. The graph above, for example, shows how a version of consensus has better availability and faster convergence time when the protocol makes use of the plausible failure patterns of a wide area network, even when no failures occur. Finally, we have developed a better understanding of the performance of core protocols in real*

Impact on the warfighter: *Failures of an information system in a real military deployment are complex and not easily characterized using the simple failure models commonly used in high-level protocol design before our work. Using the simpler models results in solutions that are slower and require more infrastructure, both of which pose problems in a deployment. With our models and new versions of basic protocols, it should be possible to build more efficient and robust information system.*

References:

Synchronous Consensus for Dependent Process Failures, Flavio Junqueira and Keith Marzullo, Proceedings of the International Conference on Distributed Computing Systems (ICDCS), Providence, Rhode Island, May 2003.

The Virtue of Dependent Failures in Multi-site Systems, Flavio Junqueira and Keith Marzullo, Proceedings of the IEEE Workshop on Hot Topics in System Dependability (HotDep), Yokohama, Japan, June 2005.

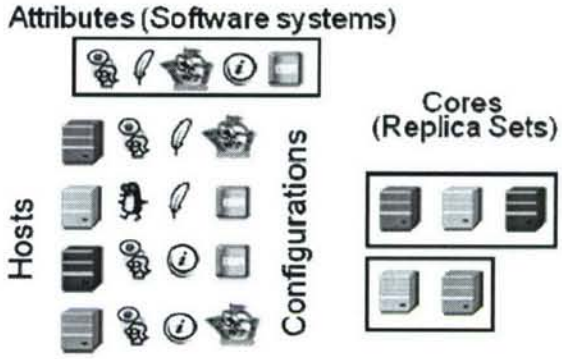
Replication predicates for dependent-failure algorithms, Flavio Junqueira and Keith Marzullo, Proceedings of the 11th International Conference on Parallel and Distributed Computing (EuroPar), August 2005.

Coterie Availability in Sites, Flavio Junqueira and Keith Marzullo, Proceedings of the International Symposium on Distributed Computing (DISC), Cracow, Poland, September 2005.

Classic Paxos vs. Fast Paxos: *Caveat Emptor*, Flavio Junqueira, Yanhua Mao and Keith Marzullo. Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07), Edinburgh, UK, June 2007.

Informed Replication

Goal: *Informed replication is an application of our dependent failure models to real distributed systems. Large-scale distributed systems are highly vulnerable to Internet catastrophes: events in which an exceptionally successful network pathogen, like a worm or email virus, causes data loss on a significant percentage of the machines connected to the network. Informed replication takes advantage of software heterogeneity among nodes in a system to efficiently and reliably ensure that replicated data and services survive.*



Approach: *The key observation that makes informed replication both feasible and practical is that Internet epidemics exploit shared vulnerabilities. By replicating a system service on hosts that do not have the same vulnerabilities, a pathogen that exploits one or more vulnerabilities cannot cause all replicas to fail. For example, to prevent a distributed system from failing due to a pathogen that exploits vulnerabilities in Web servers, the system can place replicas on hosts running different Web server software.*

Accomplishments:

- *Developed a system model using our core abstraction to represent failure correlation in distributed systems. A core is a reliable minimal subset of components such that the probability of having all hosts in a core failing is negligible.*
- *Measured and characterized the diversity of the operating systems and network services of hosts in the UCSD network to evaluate the degree of software heterogeneity found in an Internet setting.*
- *Developed heuristics for computing cores that provide excellent reliability guarantees, have low overhead, bound the number of replica copies stored by any host, and the heuristics lend themselves to a fully distributed implementation for scalability.*
- *To demonstrate the feasibility and utility of our approach, we applied informed replication to the design and implementation of the Phoenix cooperative, distributed remote backup system.*

Impact on the warfighter: *Informed replication provides a powerful approach for enabling large-scale distributed systems to survive virulent, self-propagating Internet malware. Systems that include hosts with different software configurations can take advantage of the approach, including systems deployed in the battlefield as well as military and agency IT infrastructure.*

References:

The Phoenix Recovery System: Rebuilding from the Ashes of an Internet Catastrophe, Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker, Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, HI, May 2003, pages 73-78.

Surviving Internet Catastrophes, Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker, Proceedings of the USENIX Annual Technical Conference, Anaheim, CA, April 2005, pages 45-60.

4. Personnel Supported

Faculty, Researchers, and PostDocs

Ken Birman (Cornell), Alan Demers (Cornell), Richard Eaton (Cornell), Paul Francis (Cornell), Johannes Gehrke (Cornell), Christoph Koch (Cornell), Keith Marzullo (UCSD), Jayavel Shanmugasundaram (Cornell), Niki Trigoni (Cornell), Robbert Van Renesse (Cornell), Mark Riedwald (Cornell), Geoffrey M. Voelker (UCSD), and Werner Vogels (Cornell), Walker White (Cornell).

Graduate Students

Jeanne Albrecht (UCSD), David S. Anderson (UCSD), Ben Atkin (Cornell), Alvin AuYoung (UCSD), Anand Balachandran (UCSD), Mahesh Balakrishnan (Cornell), Hitesh Ballani (Cornell), Rimon Barr (Cornell), Leeann Bent (UCSD), Ranjita Bhagwan (UCSD), Adrian Bozdog (Cornell), Cristian Bucila (Cornell), John Calandrino (Cornell), Zhiyuan Chen (Cornell), Sigmund Cherem (Cornell), Jessica Chiang (UCSD), Adina Crainiceanu (Cornell), Annemarie Dahm (UCSD), Abhinandan Das (Cornell), Chris Fleizach (UCSD), Flavio Junqueira (UCSD), Lin Guo (Cornell), Indranil Gupta (Cornell), Maya Hardisasan (Cornell), Mingsheng Hong (Cornell), Ken Hopkinson (Cornell), Tibor Janosi (Cornell), Prakesh Linga (Cornell), Ashwin Machanavajjala (Cornell), Priya Mahadevan (UCSD), Yanhua Mao (UCSD), Marvin McNett (UCSD), Alper Mizrak (UCSD), Krzysztof Ostrowski (Cornell), Biswanath Panda (Cornell), Ishwar Ramani (UCSD), Venugopala Ramasubramanian (Cornell), Manpreet Singh (Cornell), Michael Sirivianos (UCSD), Kiran Tati (UCSD), Ruijie Wang (Cornell), Ming Woo-Kawaguchi (UCSD), Dmitrii Zagorodnov (UCSD), Xianan Zhang (UCSD), Xinyang Zhang (Cornell).

Undergraduate Students

Justin Koser (Cornell), Ben Kraft (Cornell), Amar Phanishayee (Cornell).

Interactions/Transitions

The members of the team have been very active, speaking at a wide range of conferences and workshops during the course of the MURI project. In particular, Professor Birman maintains active dialog and research collaborations with many companies, and also advises the US military and government in many capacities. During the period of the MURI effort he has had active dialog with the following companies and US government agencies. Obviously, some of these have been more substantive than others, but as a group, they illustrate the extent of ties between Birman's effort and industry and support his view that as technologies emerge suitable for potential technology transition, there will be good opportunities for pursuing them further. For example, at the current time, his group is working to explore a transition path for the Ricochet technology into Raytheon's DDS platform for military publish-subscribe applications that need real-time responsiveness.

- Air Force Office of the CIO, Air Force Research Laboratories, Apache Group, Amazon, Cisco, DARPA, DHS, Microsoft, Infosys, IBM, Intel, Google, Mitre, NSF, OSD / DDR&E, OSD / DDS&T, RAND, Raytheon, SRI, White House OSTP, WSO2

Honors and Awards

Ken Birman is a Fellow of the ACM.

Johannes Gehrke received an Alfred P. Sloan Fellowship (2003).

Geoffrey M. Voelker received the UCSD Hellman Faculty Fellowship (2002), the UCSD Chancellor's Associates Award for Excellence in Undergraduate Teaching (2006), and the UCSD Jacobs School Ericsson Distinguished Scholarship (2007).

5. Key Research Publications

Finally, we include key research publications from each of the projects summarized above to provide depth and detail of our MURI research efforts.

Extensible Web Services Architecture for Notification in Large-Scale Systems

Krzysztof Ostrowski
Cornell University
krzys@cs.cornell.edu

Ken Birman
Cornell University
ken@cs.cornell.edu

Abstract

Existing web services notification and eventing standards are useful in many applications, but they have serious limitations precluding large-scale deployments: it is impossible to use IP multicast or for recipients to forward messages to others and scalable notification trees must be setup manually. We propose[1] a design free of such limitations that could serve as a basis for extending or complementing these standards. The approach emerges from our prior work on QSM [1]*, a new web services eventing platform that can scale to extremely large environments.*

1. Introduction

1.1. Motivation

Notification is a valuable, widely used primitive for designing distributed systems. The growing popularity of RSS feeds and similar technologies shows that this is also true at Internet scales. The WS-Notification [2] and WS-Eventing [3] standards have been offered as a basis for interoperation of heterogeneous systems deployed across the Internet. Unlike RSS, they are subscription-based, and hence free of the scalability issues of polling, and support proxy nodes that can be used to build scalable notification trees. Nonetheless, they embody restrictions:

- *Not self-organizing.* While both standards permit the construction of notification trees, such trees must be manually configured and require the use of dedicated infrastructure nodes ("proxies"). Automated setup of dissemination trees, formed by recipients, and without the dedicated infrastructure, is more appropriate.
- *Inability to use external multicast frameworks.* Both standards leave it entirely to the recipients to prepare their communication endpoints for message delivery. This makes it impossible for a group of recipients to dynamically agree upon a shared IP multicast address, or to construct an overlay multicast within a segment of the network. Yet such techniques are central to achieving high performance and scalability, and could also be used to provide QoS guarantees or leverage the emergent technologies.
- *No forwarding among recipients.* Many content distribution schemes build overlays within which content recipients participate in message delivery. In current web services notification standards, however, recipients are *passive* (limited to data reception).
- *Difficult to manage.* At Internet scales, it is hard to create and maintain a dissemination structure that would permit any node to serve as a publisher or subscriber, for this requires many parties to maintain common infrastructure, agree on standards, topology and other factors. Any such large-scale infrastructure should respect local autonomy, whereby the owner of a portion of a network can set up policies for local routing, availability of IP multicast, etc.
- *Weak reliability.* Reliability in the existing schemes is limited to per-link guarantees, resulting from the use of TCP. In many situations, stronger guarantees are required, e.g. to support virtually synchronous, transactional or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages or participate in multi-party protocols, even weak guarantees cannot be provided.

1.2. Our Contribution

In this document, we propose a principled approach to building large-scale systems for web services notification. We outline a design for an extensible notification scheme free of the limitations just described, which is the basis for Quicksilver [1], a new scalable and reliable publish-subscribe and notification platform under development at Cornell. Motivated by the end-to-end principle, we separate the implementation of loss recovery and strong reliability properties from the unreliable dissemination of messages. Accordingly, our design includes a *reliability framework* and a *dissemination framework*: two largely independent, yet complementary structures.

Both frameworks reflect the principles articulated below, and they share many elements. In particular, both employ hierarchical protocol stacks, an idea that is central to our architecture. These stacks permit the definition of an Internet-scale loss recovery scheme which can employ different recovery policies within different administrative domains. Likewise, they permit a construction of a global dissemination scheme that uses different mechanisms to distribute data within different administrative domains.

[1] Our effort is supported by AFRL/Cornell Information Assurance Institute.

1.3. Design Principles

The limitations of the existing designs listed above and our experience designing scalable multicast systems led us to the following design principles:

- *Programmable nodes*. Senders and recipients should not be limited to sending or receiving. They should be able to perform certain basic operations on data streams, such as forwarding or annotating data with information to be used by other peers, in support of local *forwarding policies*. The latter must be expressive enough to support protocols used in today's content delivery networks, such as overlay trees, rings, mesh structures, gossip, link multiplexing, or delivery along redundant paths.
- *External control*. Forwarding policies used by nodes must be selected and updated in a consistent manner. A node cannot predict a-priori what policy to use, or which other nodes to peer with; it must permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering etc.
- *Hierarchical structure*. The principles listed above should apply to not just individual nodes, but also entire administrative domains such as LANs, data centers or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating the cooperation among organizations in maintaining the global infrastructure. The way a message is delivered to subscribers across the Internet thus reflects policies defined at various levels.
- *Isolation and local autonomy*. A certain degree of local autonomy of the administrative domains must be preserved; such as how messages are forwarded internally, which nodes create which communication endpoints etc. In essence, the structure of a domain should be hidden from other domains it is peering with and from the higher layers. Likewise, details of its own subcomponents should as opaque as possible.
- *Channel negotiation*. Communication channel creation should permit a handshake. A recipient might be asked to e.g. join an IP multicast group, or subscribe to an external system. The recipient could then make a configuration decision on the basis of the information about the sender, e.g. a LAN asked to prepare a communication endpoint for receiving may choose a well-provisioned node to handle the anticipated load.
- *Managed channels*. Communication channels should be represented as *active contracts* in which receivers have a degree of control over the way the senders are sending. In self-organizing systems, reconfiguration triggered by churn is common and communication channels often need to be reopened or updated to adapt to the changing topology, traffic patterns or capacities. For example, a channel that previously requested that a given source transmits messages to one node, might notify the source that messages should now be transmitted to two other nodes, instead.
- *Reusability*. It should be possible to specify a policy for message forwarding or loss recovery in a standard way and post it into an online library of such policies as a contribution to the community. Administrators willing to deploy a given policy within their administrative domain should be able to do so in a simple way, e.g. by drag-and-drop, within a suitable GUI.

1.4. Basic Concepts

We employ the usual terminology, where notifications are associated with *topics* and produced by *publishers* and delivered to *subscribers*. We use the term "*group X*" to refer to the group of nodes subscribed to topic "*X*". More than one publisher may exist for a given topic. The prospective publishers and subscribers register with a *subscription manager*, which can be decentralized and independent of the publishers (see Figure 1, top). In our architecture, nodes reside in *administrative domains*. Nodes in the same domain are jointly managed. It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries; either for administrative reasons, or because communication locally in a domain is cheaper than across domains, as it is often related to network topology. Publishers and subscribers may be scattered across organizations, which must cooperate in message delivery. This often presents a logistic challenge (see Figure 1, bottom).

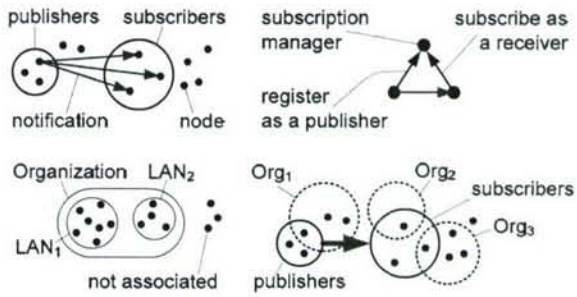


Figure 1. Publishers and subscribers register with the *subscription manager* (top). Nodes are scattered across *administrative domains* hierarchically divided into *sub-domains* (bottom).

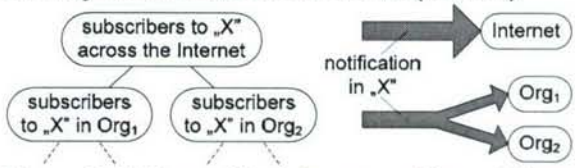


Figure 2. A hierarchical decomposition of the set of subscribers along the domain boundaries.

1.5. A Hierarchical View of the Network

A group X of subscribers for a given topic across the Internet can be divided into subsets Y_1, Y_2, ..., Y_N of subscribers in N top-level administrative domains (Figure 2). This can be continued recursively, leading to a hierarchical perspective on the set of all subscribers. By the principle of *isolation* and *local autonomy*, each administrative domain should manage the registration of its own publishers and subscribers internally, and decide how to distribute messages among them according to its local policy. Similar ideas were previously exploited in the context of content-based filtering [6], and in many scalable multicast algorithms, e.g. in RMTP [4]. This also reflects the principle of locality, implicit in many scalable protocols. Following this principle, groups of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection etc. Each such group is treated as a single cell within a global infrastructure. A protocol running at a global level connects all cells into a single structure. Scalability arises as in the *divide and conquer* principle. Additionally, the cells can locally share workload and amortize overhead, e.g. buffer messages coming from different sources and locally disseminate such combined bundles etc. We make heavy use of this property in our QSM [1] system.

This principle of locality and the hierarchical view of the network outlined above form the basis for our design.

2. Design Overview

2.1. The Hierarchy of Scopes

Our design is constructed upon the following principal concepts: *management scope*, *channel*, *filter*, *forwarding policy*, *session*, *recovery algorithm*, and *recovery domain*.

A *management scope (*or simply a *scope)* represents a *set of jointly managed nodes*. It may include a single node, span over a group of nodes residing within a certain administrative domain, or include nodes clustered based on other criteria, such as common interest. In the extreme, a scope may span the Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but it will often be the case, and we will refer to a *LAN scope* (or just a *LAN*) to mean the scope spanning over all nodes residing within a LAN. The reader might find it easier to understand our design with such examples in mind.

A scope is not just any group of nodes, the assumption that they are *jointly managed* is essential. The existence of a scope is dependent upon the existence of infrastructure that maintains its membership and administers it. For a scope that corresponds to a LAN, this could be a server managing all local nodes. In a domain that spans several data centers in an organization, it could be a management infrastructure with a server in the company headquarters indirectly managing the network via subordinate servers in data centers. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the global scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many independently managed global scopes could thus co-exist.

Like administrative domains, scopes form a hierarchy, defined by a relation of *membership*: a scope may *declare* itself to be a *member* (*sub-scope*) of another scope. If X declares itself to be a member of Y, it means X is either physically or logically a part (or subset) of Y. Typically a scope defined for a sub-domain X of some administrative domain Y will be a member of the scope defined for Y. For instance, a node would be a member of a LAN. The LAN would be a member of a data center, which in turn would be a member of a corporate network etc. A node could also be a member of a scope of some overlay network. For a data center, two scopes might be defined, e.g. a monitoring scope and a control scope, both scopes covering the entire data center, with some LANs being a part of one scope, the other scope, or both. The corporate scope could be a member of several Internet-wide scopes.

The generality in these definitions allows us to model various special cases, such as clustering of nodes based on interest or other factors. Such clusters, formed e.g. by a server managing a LAN and based on node subscription patterns, could also be considered scopes, all managed by the same server. Nodes would be members of clusters and clusters (not nodes) would be members of the LAN. As it will be explained below, each cluster, as a separate scope, could be locally and independently managed. In [1], such construction is a basis for our scalable multicast protocol.

A scope hierarchy is not a tree. There may be multiple global scopes, or many super-scopes for any given scope. However, a scope always decomposes into a tree of subscopes, down to the level of nodes. We refer to a *span of a scope X* as the set of all nodes at the bottom of a hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, i.e. such that all subscribers to X reside in the span of X. Publishing a message in a topic is thus equivalent to delivering it to all subscribers in the span of some global scope, which may be further decomposed into subscribers in the spans of sub-scopes (compare section 1.5 and Figure 2).

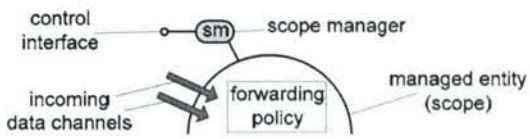


Figure 3. Accessed via a control interface and configured with a forwarding policy, a scope manager creates incoming data channels.

2.2. The Anatomy of a Scope

The infrastructure managing a scope is referred to as a *scope manager* (SM). A single SM may control multiple scopes. It may be hosted on a single node, or on a set of nodes, perhaps outside of the scope it controls. It exposes a *control interface*, a web service hosted at a well-known address, to dispatch control requests directed to scopes it controls (Figure 3). SMs interact by calling each other's control web interfaces (see also [8]).

A scope maintains communication *channels* for use by other scopes. A *channel* is a mechanism through which a message can be delivered to all those nodes in the span of this scope that subscribed to any of a certain set of topics. In a scope spanning over a single node, a channel may be just an address/protocol pair; creating it would mean arranging for a local process to open a socket. In a distributed scope, a channel could be an IP multicast address; creating it would require all local nodes to listen at this address. In an overlay network, a channel could lead to nodes that forward messages across the entire overlay.

A scope that spans over a set of nodes is governed by forwarding *policy* specifying how messages that originate within that scope or arrive through some communication channel are forwarded internally and to other scopes.

The reader will recognize in our construction the principles we formulated earlier. Scopes, whether individual nodes, LANs or overlays, are *externally controlled* using their control interfaces, may be *programmed* with policies that govern the way messages are distributed internally and forwarded to other scopes, and transmit messages via *managed* communication channels established through a dialogue between a pair of SMs.

2.3. Hierarchical Composition of Policies

Following our design principles, we propose to solve the issue of a large-scale global cooperation in message delivery between independently managed administrative domains by introducing a hierarchical structure, in which forwarding policies defined at various levels are merged into a single dissemination scheme. Each scope is configured with a policy dictating, on a per-topic (and perhaps a per-sender) basis, how messages are forwarded among its members. For example, a policy governing a global scope might determine how messages in topic T, originating in a corporate network X, are forwarded between the various organizations. A policy of a scope of the given organization's network might determine how to forward messages among its data centers, and so on. A policy defined for a particular scope X is always defined at the granularity of X's members (not individual nodes). The way a given sub-scope Y of X delivers messages internally is a decision made by Y autonomously. Similarly, X's policy may specify that Y should forward messages to Z, but it is up to Y's policy to determine how to perform this task.

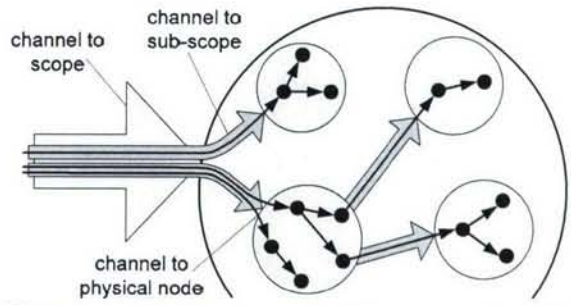


Figure 4. Channels created in support of forwarding policies defined at different levels.

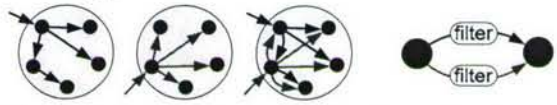


Figure 5. Forwarding graphs for different topics are superimposed. Two members may be linked by multiple channels, each with a different filter.

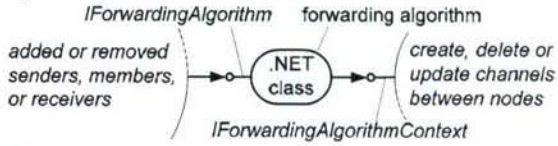


Figure 6. A forwarding policy as a code snippet.

Accordingly, a global policy may request organization X to forward messages in topic T to organizations Y and Z. A policy governing X may then determine that to distribute messages in X, they must be sent to LAN_1, which will forward them to LAN_2. The same policy might also specify which LANs within X should forward to Y and Z. Finally, the policies of the respective LANs could delegate these tasks to individual nodes. When the policies defined at all the scopes involved are combined, the resulting *forwarding structure* completely determines the way messages are forwarded (see Figure 4).

In the examples above, the policies are simply graphs of connections: each message is always forwarded along every channel. In general, however, each channel may be optionally constrained by a *filter* that decides, on a per-message basis, whether to forward or not, and optionally tags the message with custom attributes. This allows us to express many popular techniques, e.g. using redundant paths, multiplexing between dissemination trees etc.

Every scope manager maintains internally a mapping from topics to policies: a graph of channels to create and filters on them. Such graphs of connections for different topics are superimposed (see Figure 5). Based on this, the SM asks the scope members to create channels and filters. When the structure is modified as a result of membership or subscription changes, the SM makes additional control requests to reflect this.

In our framework, a policy is defined as an algorithm that lives in an *abstract context*, with a fixed set of *events*

to respond to, standard set of *operations* and *attributes* to inspect. In a prototype we are developing, we implement a forwarding policy as a .NET class, stored in a DLL on an *algorithm library* server, that implements an abstract interface and interacts with an abstract *context* hiding the details of the environment (Figure 6). This allows our policies to be used within any scope.

2.4. Communication Channels

Consider a node X, a member of a scope Z that, based on a forwarding policy at Z, has been requested to establish a communication channel to scope Y to forward messages in topic T. Following the protocol, X asks the SM of Y for the specification of the channel to Y that should be used for messages in topic T. The SM of Y might respond with an address/protocol pair that X should use to send over this channel. Alternatively, a forwarding policy defined for T at scope Y may dictate that, in order to send to Y in topic T, X should establish channels to members A and B of Y, constrained with filters α and β. After X learns this from the SM of Y, it contacts SMs of A and B for details. Notice how the channel decomposes into sub-channels to A and B through a policy at a target scope Y.

This decomposition continues hierarchically, until the point when scope X is left with a tree containing filters in internal nodes and address/protocol pairs at the leaves (Figure 9). In order to send a message along the channel built in this way, X executes filters to determine which sub-channels to use, proceeding recursively, until it is left with a list of address/protocol pairs, then transmits the message. Filters will typically be simple, such as modulo-**n**; hence X could perform this procedure very efficiently.

Accordingly, to support the hierarchical composition of policies described in the preceding section, we define a channel as one of the following: an address/protocol pair, a reference to an external multicast mechanism, or a set of sub-channels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which sub-channels to use for sending, on a per-message basis (see Figure 7 and Figure 8).

Consider now the case when scope X, spanning over a set of nodes, has been requested to create a channel to scope Y. Through a dialogue with Y and its sub-scopes, X can get a detailed channel definition, but unlike in the example above, X now spans over a set of nodes, and as such, it cannot *execute* filters or *send* messages.

We propose two example generic techniques that solve this problem: *delegation* and *replication* (Figure 10). Both rely on the fact that if scope X receives messages in a topic T, then some of its members, Z, must receive them (for otherwise X would not be made part of a forwarding structure for topic T by X's super-scope). In case of *delegation*, X requests such a sub-scope Z to create the channel on behalf of X, essentially delegating the whole channel. The problem can be recursively delegated, down to the level where a single physical node is requested to create a channel. A more sophisticated use of delegation would be for X to delegate sub-channels. In such case, X would first contact Y to obtain the list of sub-channels and the corresponding filters, and for each of these sub-channels, delegate it to one of its sub-scopes. In any case, X delegates the responsibility for sending over a channel, in one way or another, to one or more of its sub-scopes.

In case of *replication*, scope X requests **n** of its sub-scopes to create the channel, but constrains each with a modulo-**n** filter based on a message sequence number (i.e. sub-scope **k** only forwards messages with numbers **m** such that **m** mod **n** equals **k**), effectively implementing a round-robin policy. Although all sub-scopes would create the same channel, the round-robin filtering policy ensures that every message is forwarded only by one of them.

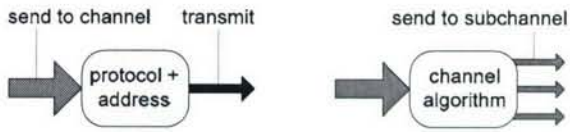


Figure 7. A channel may be an *address/protocol* pair (left), or it may consist of *sub-channels*, with an *algorithm* deciding what goes where (right).

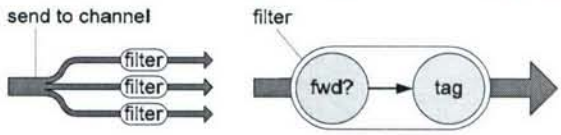


Figure 8. Channel algorithms are realized as sets of filters, one per subchannel, deciding whether to forward, and optionally adding custom tags.

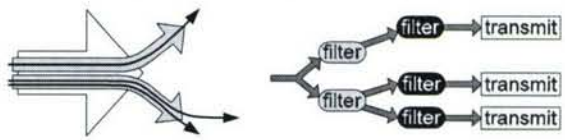


Figure 9. A channel split into sub-channels and a possible filter tree corresponding to it.

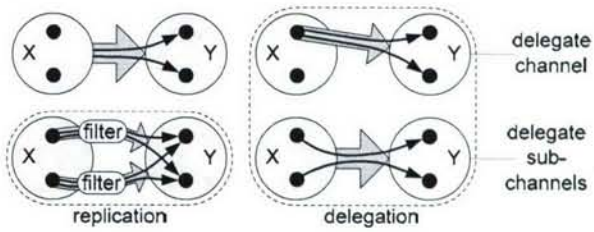


Figure 10. A distributed scope may delegate a channel or its sub-channels to members, or it may replicate them among members with filters that jointly implement a round-robin policy.

2.5. Reliability Scopes

The design of the reliability framework also relies on the concept of management scopes, referred to here as *reliability scopes* (in contrast to *dissemination scopes* in the dissemination framework). A reliability scope isolates and encapsulates the local aspects of loss recovery, hiding details from other scopes, just like a dissemination scope hides the local aspects of message delivery. Reliability scopes are also controlled by scope managers. Both kinds of scopes would typically overlap. For example, a single scope could be defined for an administrative domain such as a LAN, isolating local aspect of both dissemination and reliability. The scope could then be controlled by a single SM managing both dissemination and reliability.

The separation of dissemination from reliability makes it possible to combine an arbitrary unreliable notification mechanism, such as IP multicast or an overlay content delivery system, with a wide range of reliability protocols expressible in our reliability framework. This degree of reusability has not been possible with prior architectures.

2.6. Hierarchical Approach to Reliability

Our approach to reliability resembles our approach to dissemination. Just as channels are decomposed into sub-channels, in the reliability framework we decompose the task of repairing after message losses and providing other reliability goals. Recovering messages in a certain scope is modeled as recovering within sub-scopes, and then recovering "among" the sub-scopes (Figure 11). Just like recovery among single nodes, recovery among LANs may involve comparing their "state" (such as aggregated ACK or NAK information for the entire LANs) and forwarding lost messages. In section 2.9 we give examples of how recovery protocols may be defined and combined.

In our framework, different recovery schemes may be used in different scopes, to reflect differences in network topology, node or network capacity, the way subscribers are distributed (e.g. clustered vs. scattered around) etc.

Just like messages are disseminated through channels, reliability is achieved via *recovery domains*. A recovery domain **D** in scope **X** may be thought of as a "distributed recovery protocol running among some nodes in **X**" that performs recovery-related tasks for a certain set of topics. The concept of a recovery domain is symmetric, dual to the notion of a channel. We present it via analogy.

- Just like a channel is created to disseminate messages for some topics $\mathbf{T_1}$, $\mathbf{T_2}$, ... $\mathbf{T_k}$ in scope **X**, a recovery domain is created to handle loss recovery and other reliability tasks, again for a specific set of topics and in a specific scope. Just like there may be multiple channels to a scope, e.g. for different sets of topics, multiple recovery domains, each for different topics, may exist within a single reliability scope.
- Just like channels may be composed of sub-channels, a recovery domain **D** defined at scope **X** may be composed of *sub-domains* $\mathbf{D_1}$, $\mathbf{D_2}$, ... $\mathbf{D_n}$ defined at sub-scopes of **X** (we will call them *members* of D). Each such sub-domain $\mathbf{D_i}$ handles recovery for a certain set of subscribers in the respective sub-scope, while **D** handles recovery "across" its sub-domains.
- Just like channels are composed of sub-channels via applying filters assigned by forwarding policies, a recovery domain **D** performs its recovery tasks using a *recovery algorithm*. Such an algorithm, assigned to **D**, specifies how to combine recovery mechanisms in the sub-domains of **D** into a mechanism for all of **D**. Recovery algorithms are defined in terms of how the sub-domains "interact" with each other. We will see how this is achieved in section 2.9.
- Just like a single channel may be used to disseminate messages in multiple topics, a recovery domain may run a single protocol to perform recovery for multiple topics. In both cases, reusing a single mechanism (a channel, a token ring etc.) may significantly improve performance due to the reduction in the total number of "control" messages. We evaluated this idea in [1].

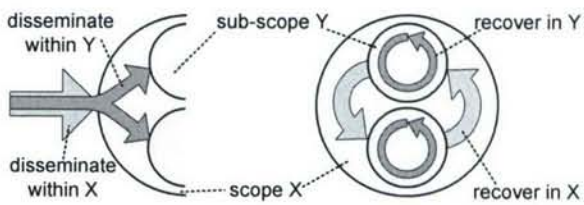


Figure 11. The similarity between hierarchical dissemination (left) and recovery (right).

Each individual node is a recovery domain on its own. In a distributed scope such as a LAN, on the other hand, two cases are possible. First, a single domain may cover the entire LAN. All internal nodes could form e.g. a token ring, exchange ACKs for messages in all topics, and use this to arrange for local repairs. Another possibility is that separate domains would be created for every individual topic. Subscribers to different topics would form separate structures, such as ring or trees, and run separate protocol instances in each, exchanging state and los messages.

As explained later, recovery domains in our system actually handle recovery for specific *sessions*, not just for specific topics. Sessions are introduced in section 2.7.

A recovery domain D of a data center could have as its members recovery domains created in LANs. Note that in this case, members of D would be sets of nodes. A recovery algorithm running in D would specify how all these different sets of nodes should exchange state and forward lost messages to one another. Note the similarity to a forwarding policy in a data center, which would also specify

how messages are forwarded among sets of nodes. As shown in section 2.10, recovery algorithms are implemented through delegation, just like forwarding. A concept of a *recovery algorithm* is, to a certain extent, symmetric to the notion of a forwarding policy.

2.7. Sessions

Within our architecture, protocols that provide strong reliability guarantees express them in terms of *epochs*. An epoch corresponds to what in group communication literature is called a *membership view*. The lifetime of a topic is divided into a sequence of epochs. Whenever the set of subscribers to a topic changes as a result of a subscribe/unsubscribe request or a failure, the event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs. One then defines consistency in terms of which messages may be delivered to which subscribers and at what time, relative to epoch boundaries. The traditional term "membership view" reflects the fact that epochs begin and end with membership change events. The set of subscribers during a given epoch is fixed.

Although simple protocols, such as SRM or RMTP, do not rely on a consistent view of group membership, and their properties are not defined in terms of epochs, epochs are still a useful, if not a universal concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, e.g. to rebuild a distributed structure, release resources or cancel activity that is no longer necessary. Many simple protocols simply do not take this factor into account.

We introduce the idea of a *session*, a generalization of an epoch (membership view). A session is also an epoch in the prior sense, i.e. the lifetime of any given topic can always be divided into a sequence of sessions. Like before, any membership change marks the beginning of a new session and for a given session, membership is fixed. However, a new session may also be initiated even if membership is unchanged. The reliability properties of a group may vary to some extent in the subsequent sessions. An important example is an administrative change, where a new protocol is introduced, e.g. because it is more efficient or to fix a bug in the existing protocol. In Internet-scale systems such administrative changes must be performed online; session changes achieve this.

Session numbers are assigned globally for consistency. As explained before, for a given topic, a single "global" scope always exists such that all subscribers to that topic reside within the span of this scope. This is true for both dissemination and reliability frameworks. Usually, both global scopes overlap and are managed by a single SM. The top-level SM assigns and updates session numbers. Note that local topics (e.g. internal to an organization) could be managed by the local SM, much in a way local newsgroups are visible and managed locally.

Before discussing the mechanisms used to manage membership, we conclude the discussion of sessions by explaining how they impact the behavior of publishers and subscribers. After registering, a publisher waits for the SM to notify it of the session number to use for a particular topic. A publisher is also notified of changes to the session number for topics it registered with. All published messages are tagged with the most recent session number, so that whenever a new session is started for a topic, within a short period of time no further messages will be sent in the previous session. Old sessions eventually quiesce as receivers deliver messages and the system completes flushing, cleanup and other reliability mechanisms used by the particular protocol. Similarly, after subscribing to a topic, a node does not process messages tagged as committed to session **k** until it is explicitly notified that it should receive messages in that session. Later, after session **k+1** starts, all subscribers are notified that session **k** is entering a *flushing* phase (this term originates in *virtual synchrony* protocols, but similar mechanisms are common in reliable protocols; a protocol lacking a flush mechanism simply ignores such notifications). Eventually, subscribers report that they have completed flushing and a global decision is made to cease activity and *cleanup* resources pertaining to session **k**, completing the transition.

2.8. Constructing the Recovery Structure

Reliable protocols often rely on, or could benefit from, a consistent view of membership. It helps to determine which nodes have crashed or disconnected. In existing systems, this is achieved by a Global Membership Service (GMS) that monitors failures and membership changes, decides when to install new membership views for topics, and notifies all affected members of the new views. In our framework, the global SM for a given topic is responsible for announcing when sessions begin and end. However, if the global SM had to process all subscriptions, it would lead to a non-scalable design that violates the principle of isolation. To avoid this, for each topic **T** we distribute the information about membership of **T** across all SMs in the hierarchy of scopes for **T** (this hierarchy was defined in section 2.1). Each SM thus has only a partial membership view for each session. This scheme is outlined below.

In the reliability framework, if a scope **X** subscribes to a topic **T**, it specifies some local recovery domain **D** that should handle the recovery for topic **T** in **X**. The **X**'s super-scope **Y** processes this subscription request jointly with requests from other sub-scopes. It then creates its own recovery domains, with the newly subscribed and perhaps some existing sub-domains as members, and then issues its own subscription requests to its super-scope. This continues recursively up to the global scope.

The scheme used by the super-scope to create recovery domains must abide by three rules. First, the list of sub-

domains of a recovery domain is determined once at the time of creation, and fixed throughout its lifetime. This is necessary to ensure that a hierarchical structure employed for recovery in any given session does not change, which simplifies the overall design. Second, a recovery domain **D** at scope **X** is responsible for handling recovery for a specific set of topics, in specific sessions. If a change in membership in any of these topics occurs locally in **X**, a new recovery domain **D'** must be created, and when a new session is announced, it is installed in **D'**. This is because the existing recovery domain **D** no longer represents the current set of subscribers within **X**, hence a new distributed structure **D'** must be established. Finally, if a new session is announced for some topic **T**, but no membership changes occurred for **T** within scope **X** since the previous session, then an existing recovery domain should be re-used to handle recovery in the new session.

In a scope in which recovery for each topic is handled individually, we would maintain a separate sequence of recovery domains for each topic. A new domain would be created whenever the set of subscribers locally changes. In a scope in which recovery for all topic is performed jointly, such as e.g. in a cluster of nodes defined based on subscription patterns in which all nodes are subscribers to the same set of topics, there will be just a single sequence of recovery domains. We used the latter scheme in [1].

The above procedure effectively constructs a hierarchy of sub-domains, with the property that for each topic **T**, the recovery domains subscribed to **T** form a tree.

The global scope assigns new session numbers for all topics for which subscribe or unsubscribe requests have been received, and determines which of its local recovery domains should handle the new sessions. This represents a coarse-grained membership view, for each session only top-level recovery domains are specified, with no further details. The information about the new sessions is now sent down the tree of subscribers, and transformed along the way to filter out unnecessary details. The membership information a scope **X** receives for a session **S** is limited to one level "above" **X**, i.e. it includes **X**'s own recovery domain that got subscribed to **S** and the recovery domains of its *sibling* scopes (i.e. scopes that have the same super-scope). It is also coarse-grained, i.e. it does not provide any details at the level "below" **X** or its siblings.

2.9. Modeling Recovery Algorithms

The design of the reliability framework is based on an abstract model of a distributed protocol dealing with loss recovery and other reliability properties. When expressed within our framework, such protocols will be referred to as *recovery algorithms*. Recovery algorithms are the basic building blocks in constructing our hierarchical reliability protocols, much in a way channels and filters are the basic building blocks in our forwarding infrastructure.

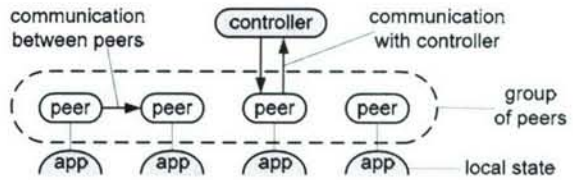


Figure 12. A group of *peers* in a reliable protocol.

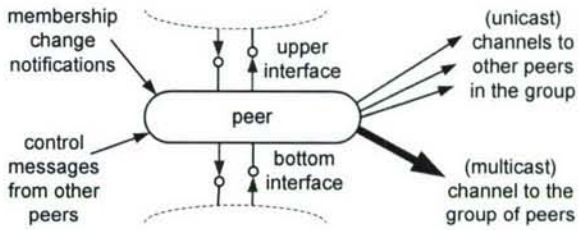


Figure 13. A peer modeled as a component living in abstract environment (events, interfaces etc.).

A protocol such as SRM, RMTP, or virtual synchrony is defined in terms of a group of cooperating *peers* that send control messages and forward lost packets to each other, and perhaps to a distinguished node, such as a sender or some node higher in a hierarchy, that we will refer to as a *controller* (Figure 12). The controller does not have to be a separate node; this function could be served by one of the peers. The distinction between the peers and the controller may be purely functional. The point is that the group of peers, as a whole, may be asked to perform a certain action, or calculate a value, for some higher-level entity, e.g. a sender, a higher-level protocol, or a layer in a hierarchical structure etc. Examples of such actions include requesting or performing a retransmission for all nodes, reporting which messages were successfully delivered to all nodes etc. Irrespectively of how exactly the interaction with a controller is realized, it is present in this form or another in almost every protocol run by a set of receivers. We shall refer to it as an *upper interface*.

Each peer inspects and controls *local state*. Such state may include e.g. a list of messages received and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered etc. Operations a peer may issue to change the local state could include e.g. retrieving/purging messages from a local cache, marking messages as "deliverable", handing a previously missed message to the application or assigning message sequence in a "totally ordered" group. We refer to such operations, used to view or control local state, as a *bottom interface*.

In protocols offering strong guarantees, peers typically know the membership of their group, received as a part of the initialization process, and subsequently updated via *membership change* events. Peers send control messages to each other to share state or to request actions, such as forwarding messages. Sometimes, as in SRM, a multicast channel to the entire peer group exists.

To summarize, in most reliable protocols a peer can be modeled as running in an environment that provides the following: a *membership view* of its peer group, *channels* to all other peers, and sometimes to the entire group, a *bottom interface* to inspect or control local state, and an *upper interface* to interact with a sender or a higher level in the hierarchy concerning the state of the whole group, (Figure 13). In some protocols, parts of the environment might be unavailable, e.g. in SRM peers might not know other peers. The bottom and upper interfaces would vary.

This model is flexible enough to capture the key ideas and features of a wide class of protocols, including virtual synchrony. However, because in our framework protocols must be reusable in different scopes, they may need to be expressed in a slightly different way, as explained below.

In the RMTP protocol [4], the sender and the receivers for a given topic form a tree. Within this tree, each subset of nodes consisting of a parent and child nodes serves as a separate, local recovery group. The child nodes in every such group send their local ACK/NAK information to the parent node, which arranges for a local recovery within the recovery group. The parent itself is either a child node in another recovery group, or it is a sender, at the root of the tree. Packet losses in this scheme are recovered on a hop-by-hop basis, top-down or bottom-up, one level at a time. This scheme distributes the burden of processing the individual ACKs/NAKs, and of retransmissions, which is normally the responsibility of the sender. This improves scalability and prevents the "ACK implosion".

There are two ways to express RMTP in our model. One approach is to view each recovery group consisting of a parent node and its child nodes as a separate group of peers (Figure 14). Since internal nodes in the RMTP tree simultaneously play two roles, a "parent" node in one recovery group and a "child" node in another, we think of a node as running two "agents", each representing a different "half" of the node and serving as a peer in a separate peer group. Every group of peers, in this perspective, includes the "bottom agent" of a parent node and "upper agents" of child nodes. When a node sends messages to its child nodes as a result of receiving a message from its parent, of vice versa, we may think of those two "agents" as interacting with each other through a certain interface that one of them views as upper, and the other as bottom. These two agents play different roles, as explained below.

The bottom agent of each node interacts via its *bottom interface* with the local state of the node. It also serves as a distinguished peer in the peer group composed of itself and the upper agents of child nodes. A protocol running in this peer group is used to exchange ACKs between child nodes and the parent node and arrange for message forwarding between peers, but also to calculate collective ACKs for the peer group, i.e. which messages were not recoverable in the group. This is communicated by the bottom agent, via its *upper interface*, to the upper agent.

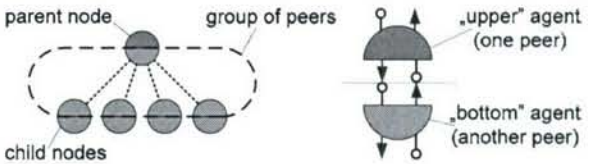


Figure 14. RMTP expressed in our model. A node hosts "agents" playing different roles.

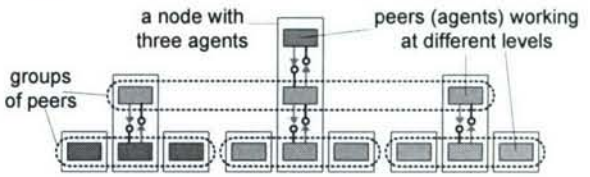


Figure 15. Another way to express RMTP. Each node hosts multiple "agents" that act as peers at different levels of the RMTP hierarchy.

The upper agent of every node interacts via its *bottom interface* with the bottom agent. What the upper agent considers as its "local state" is not the local state of the node. Instead, it is the state of the entire recovery group, including the parent and child nodes, that is collected for the upper agent by the bottom agent.

Such interactions, between a component that is a part of a "higher layer" and a component that resides in a "lower layer", both components co-located on the same physical node and connected via their upper and bottom interfaces, are the key element in our architecture.

At the top of this hierarchy, the upper agent of the root node communicates through its *upper interface* the state of the entire tree of receivers to the sender.

The second way to model RMTP captures the essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node may host multiple agents, connected to each other, each working at a different level (Figure 15). In a LAN scope, all nodes host a "local agent" component (green), similar to the "bottom agents" above, that serves as a peer in the group of all LAN nodes. The *bottom interface* used by this agent interacts with the local state. These peers exchange ACKs and arrange for message forwarding, with one of them acting as a "parent" and all other as "children". On the node hosting the "parent", a "higher-level" agent is hosted (orange); we refer to it as a "LAN agent", for there is exactly one in each LAN, and it represents the entire LAN. It connects through its bottom interface to the local agent, which is a distinguished peer in a LAN peer group, to obtain information concerning the LAN it is controlling, e.g. ACKs. These LAN agents themselves form a "higher-level" peer group. One serves as a distinguished parent node, others as subordinates. The LAN agents are communicating with each other to arrange for forwarding messages, and they jointly calculate the ACK information for the entire scope, which in

this case could be e.g. a data center in which the LANs reside. The distinguished node that hosts the parent LAN agent also hosts a yet higher-level component, call it a "data center agent". This agent could communicate with the sender, or the construction might continue further in a similar fashion. Note how in this example the peer groups defined at various levels overlap with scope boundaries.

Note also that as long as their interfaces match, each peer group could run an entirely different algorithm. We believe this power could be extremely useful in settings where local administrators control policies governing, for example, use of IP multicast and hence where different groups may need to adhere to different rules.

The issue of how to select protocols at different levels in such a way that their interfaces would match is beyond the scope of this paper. In our forthcoming paper [7], we introduce a new mechanism that could help address this issue in a more systematic manner.

To keep the presentation simple, in the model and in the examples we discussed a peer group handles recovery in a single topic. In our full design, a group of peers can handle recovery in multiple sessions at once. Throughout the lifetime of the group, peers will be instructed to begin recovery for certain sessions, at some point later they will enter the flushing phase for specific sessions (while other sessions may still be active), and may finally be requested to cease any activity for specific sessions. Accordingly, a peer, via its bottom and upper interfaces, exchanges data and requests related to multiple sessions at once. One may think of a peer as having multiple pairs of bottom and upper interfaces, each pair for a different set of sessions. Also, peers hosted at a physical node will not necessarily form a vertical, linear stack, as in our examples, the same lower-level peer may interact with two or more peers at a level above it. We omit details for clarity. All techniques that we introduced here carry over to the full design.

2.10. Implementing Recovery Algorithms

In section 2.8 we have explained how a hierarchy of recovery domains is built, such that for each session, the domains "responsible" for it form a tree. In section 2.9 we gave an example of how an algorithm such as RMTP can be modeled in our framework as a network of agents that handle the recovery tasks at various levels. A distributed recovery domain **D** in our framework will correspond to a peer group. When **D** is created at some scope **X**, the latter selects an algorithm to run in **D**, e.g. a ring or a tree, and then every sub-domain $\mathbf{D_k}$ of **D** is requested to create an agent that acts as a "peer $\mathbf{D_k}$ in group **D**". Note how the membership algorithm provides membership view at one level "above", i.e. the scope that owns a particular domain would learn about domains in all the sibling scopes. This is precisely what is required for each peer $\mathbf{D_k}$ in a group **D** to learn the membership of its group.

When the SM of a scope **X** learns that an agent should be created for one of its recovery domains $\mathbf{D_k}$ in group **D**, two things may happen. If **X** manages a single node, the agent is created locally. Otherwise, **X** delegates the task to one of its sub-scopes. As a result, the agents that serve as "peers" at the various levels are delegated to individual nodes. We thus arrive at a structure just like on Figure 18, where each node has a stack of one or more agents, each operating at a different level, linked to one another. When the node hosting a "higher-level" agent crashes, the agent is delegated to another node. Since our framework would transparently recreate channels between agents, it looks to other peers agents as if the agent lost its cached state (not permanently, for it can still query its bottom interface and talk to its peers). This requires that algorithms be defined in a way allowing peers to crash and resume with some of their state erased. Based on our experience, for a wide class of protocols this is not hard to achieve.

3. Evaluation

The need for brevity precludes a detailed discussion of the performance of our architecture. The strength of this design lies in its extensibility, ability to accommodate a wide range of transport and recovery protocols, and in facilitating the cooperation among independent parties in creating a global publish-subscribe infrastructure. Such benefits are hard to quantify. However, in certain scenarios, our approach can also greatly improve scalability. In [8], we show how we used the model and principles presented here as the basis for the design of QSM [1], a new publish-subscribe platform offering a simple ACK-based reliability and extremely scalable in multiple dimensions. We are also in the process of creating a reference implementation of the infrastructure outlined here. Ultimately, this effort will lead to a set of specifications similar to [2].

5. References

[1] K. Ostrowski, K. Birman, and A. Phanishayee, "QuickSilver Scalable Multicast". In submission, 2006.

[2] http://ifr.sap.com/ws-notification/ws-notification.pdf

[3] http://ftpna2.bea.com/pub/downloads/WS-Eventing

[4] S. Paul, and K. Sabnani, "Reliable Multicast Transport Protocol". Journal of Selected Areas in Communications (1997).

[5] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing". IEEE/ACM TONS (1996).

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". ICDCS '99.

[7] K. Ostrowski, K. Birman, "Achieving Modularity and Scalability via Typed Communication Endpoints". Forthcoming.

[8] K. Ostrowski, K. Birman, "Extensible Web Services Architecture for Notification in Large-Scale Systems (Extended Version)". Cornell University Technical Report. Forthcoming.

Exploiting Gossip for Self-Management in Scalable Event Notification Systems

Ken Birman, Anne-Marie Kermarrec, Krzystof Ostrowski,
Marin Bertier, Danny Dolev, Robbert Van Renesse
Cornell University, Ithaca; INRIA/IRISA and IRISA/INSA, Rennes; Hebrew University, Jerusalem

Abstract

Challenges of scale have limited the development of event notification systems with strong properties, despite the urgent demand for consistency, reliability, security, and other guarantees in applications developed for sensitive tasks in large enterprises. These issues are the focus of Quicksilver, a new multicast platform targeted to large-scale deployments. An initial version of the system can support large numbers of overlapping multicast groups, high data rates and groups with large numbers of members. However, Quicksilver still requires manual help when discovering the system configuration and can't easily enforce certain types of application monitoring and integrity constraints. In this paper, we propose to extend Quicksilver by introducing gossip mechanisms, yielding a self-managed event notification platform. The two technologies are presented through a single interface and appear to end users as live distributed objects, *side-by-side with other kinds of typed components.*

1. Introduction

As we look to the next generation of distributed computing platforms, it is hard not to feel concern at the accelerating deployment of systems that will play sensitive roles, and yet will be built using fragile technologies. For example, an electronic health records system must achieve high levels of availability and consistency, be largely self-configuring, and maintain privacy and security. A typical deployment scenario involves decentralized systems linked over networks, integrating subsystems running at hospitals, other care providers, laboratories, insurance companies, pharmacies, etc. Electronic monitoring devices and other sensors running both in the hospital and at home will contribute time-sensitive data, and some therapeutic and drug delivery devices will be remotely controlled.

To reduce cost and leverage standardization, a system of this sort would probably be constructed using COTS platform technologies, such as web services. Doing so also brings productivity benefits, in the form of development tools and runtime support, and makes it easy to integrate pre-supplied functions with new application-specific ones. However, today's solutions lack the sorts of strong properties needed for sensitive uses. Our objective is to extend these platforms by adding robust tools that bridge gaps while complying with standards.

The nerve center of a modern service-oriented architecture is its event notification subsystem. Event notification services can distribute sensor readings and other kinds of updates to widely distributed system components, and can be used to replicate information where an application or a record is available at multiple locations. By decoupling publishers from subscribers, these services make it easy to upgrade an application over time and to integrate components that run on dissimilar platforms or were implemented using very different technologies. On the other hand, traditional event notification platforms lack the strong guarantees needed for medical decision making and other critical roles.

If we can create a new kind of scalable, robust event notification architecture that fits seamlessly into modern development platforms such as Windows .net or J2EE, and yet has strong properties that reduce to rigorously specified protocols that the end user can count upon and reason about, we can help application developers create robust applications for sensitive uses.

In this paper, we focus on scalability, robustness and self-management, deferring issues of security and privacy for the future. For scalable event notification with strong reliability guarantees, we've developed Quicksilver: a high-performance multicast technology that can implement a variety of reliability models, including consensus-based ones [1][2]. Traditionally, systems implementing reliable multicast have scaled poorly, but as reported below, this problem can be overcome. Moreover, although we don't tackle the question here, we believe that Quicksilver can be secured using digital certification certificates, by authenticating access to information resources, and encrypting all network traffic using per-event-channel keys that can be refreshed whenever the set of subscribers changes.

The existing version of Quicksilver is weaker with respect to self-configuration and self-management; both critical requirements for the sorts of applications we hope to support. In our target environments, the pace of reconfiguration could be very rapid: if a patient falls ill, providers might (in effect) hand the family a box full of equipment to be deployed throughout the home. Needs change as the patient's care plan evolves. Patients are moved from unit to unit. Thus one must imagine a highly dynamic, rather unpredictable environment in which the

---

[1] Contact: ken@cs.cornell.edu; The Cornell research group was supported by grants from AFRL/IFSE, AFOSR, NSF and the Intel Corporation.

sets of components, their configurations, and their communication patterns change constantly. Against this backdrop, we seek an event notification infrastructure that can configure itself, that can adapt as conditions evolve, and that can be leveraged to support self-configuring applications.

Fault-tolerance poses closely related problems. Today, Quicksilver offers fault-tolerance through models such as virtual synchrony, where applications are structured into groups and, if desired, will be notified when membership changes. But not all integrity constraints map easily to group membership tracking. For example, the decoupling of publisher from subscriber is advantageous from a development perspective, but sometimes correct function requires that there be an active subscriber associated with certain topics. One such case involves logging accesses to patient records for offline audits. If this functionality is implemented using event notification, it important that the logging service be running when audit events are published. Yet even if built upon a substrate such as Quicksilver, today's event notification APIs lack mechanisms to express such constraints, and hence can't trigger exceptions when they are violated.

To address self-* needs, both within Quicksilver and in applications built using it, we propose to use technology emerging from work on *gossip* protocols. Gossip encompasses a large class of protocols that exploit randomness to achieve surprising robustness under a wide range of operating conditions. They can be made self-configuring, adapt rapidly after disruption, and support a diversity of useful end-user functionality.

The integration of gossip with multicast in a single setting poses non-trivial systems-engineering challenges. Here, we propose such a unification. Although our new system is still under development, it will offer a seamless infrastructure in which Quicksilver runs side-by-side with gossip-based mechanisms to provide a self-managed scalable event notification capability. The system will expose these gossip mechanisms so that applications can exploit them directly in the same paradigm used to expose Quicksilver's multicast functionality. Here we sketch out the architecture and discuss some research challenges it poses; several appear to be of broader relevance.

The paper is structured as follows. First, we spend a moment discussing the strengths and limitations of gossip technologies. The goal is not to be exhaustive, but rather to identify styles of gossip that are both highly effective and well matched to our self-management objectives. Next, we review Cornell's new platform, Quicksilver, touching both on its scalability and its unusual embedding into the Windows .net framework. The latter topic emerges as a source of leverage in what we are now proposing to do. Finally, we explore the options for integrating the two, arriving at an architecture that (we believe) is interesting in several respects. First, it sets gossip side by side with scalable event notification. Next, the system offers an elegant embedding into Windows so that developers can benefit from that system's powerful component integration functionality and development tools (a Linux version is also under design). And finally, it suggests a path for future evolution of service-oriented architectures and standards. The paper concludes by discussing open research questions.

2. Gossip protocols

A *gossip protocol* is one with the following properties:

1. The core of the protocol involves periodic, pairwise, inter-process interactions.
2. The information exchanged during these interactions is of (small) bounded size.
3. When node *a* interacts with node *b*, the state of *a* evolves in a way that reflects the state of *b* (and vice versa). For example, if *a* pings *b* merely to measure RTT, this is not a gossip interaction.
4. Reliable communication is not assumed.
5. The frequency of the interactions is relatively low when compared to typical message latencies.
6. There is some form of randomness in peer selection.

There are three prevailing styles of gossip protocol.

1. *Dissemination (rumor-mongering) protocols.* These use gossip to spread information; they basically work by flooding nodes in the network, but in a manner that produces bounded worst-case loads:
   a. An *event dissemination* protocol runs in response to events and can be understood as using gossip to carry out multicasts, although the events don't actually trigger the gossip (since gossip runs periodically).
   b. A *background data dissemination* protocol gossips continuously to track the evolution of state at participating nodes.
2. *Anti-entropy protocols* repair replicated data by comparing replicas and reconciling differences.
3. *Aggregation protocols* compute a network-wide aggregate by sampling information at the nodes in the network and combining the values to arrive at a system-wide value – the number of nodes in the system, the sum or average of some value, etc

Our definitions are rather broad; indeed, many protocols that predate the earliest use of the term "gossip" fall within our definition. In particular, notice that a gossip substrate can "mimic" a standard routed network. That is, nodes could "gossip" about traditional point-to-point messages, in effect tunneling normal traffic through a gossip layer. Bandwidth permitting, this implies that a gossip system can potentially support any classic protocol or distributed service. Nonetheless, when we talk of gossip, we rarely intend such a broadly inclusive

interpretation. More typically we have in mind protocols that run in a regular, periodic, relatively lazy, symmetric and decentralized manner; the high degree of symmetry among nodes is particularly characteristic. To illustrate this point, consider that one could run a 2-phase commit protocol over a gossip substrate, piggybacking the messages on gossip traffic. In our view, doing so would be at odds with the spirit of the definition: there's nothing wrong with such a protocol, but it isn't gossip!

2.1 The Limitations of Gossip

The stylized manner in which we normally use gossip introduces significant limitations. First, consider the implications of the small, bounded message sizes and the relatively slow periodic message exchanges. These combine to limit the information carrying capacity of a gossip algorithm. For example, if gossip is used to disseminate information (often, in a form of flooding), the system-wide capacity for new events will be limited simply because the aggregate "bandwidth" available is bounded. The problem is that gossip protocols keep the nodes in a network busy while information spreads – typically, a process that requires O(log(n)) time. It follows that the "rate" at which events can be introduced will be proportional to 1/log(n).

The relatively slow spread of gossip can also be an obstacle. While it is common to claim that users need only tune the gossip rate to match their goals, requirement 5 complicates the picture. Gossip rates approaching the network RTT are out of the question.

Finally, gossip can be fragile in the face of malicious behavior (components that malfunction, for example by running the protocol incorrectly, disseminating incorrect data, and so forth). Recent work on BAR Gossip [21] tries to overcome some of the issues by using verifiable pseudo-random peer selection to avoid selfish and malicious behaviors. But this is just a first step.

2.2 Strengths of Gossip

Although gossip has limitations, these protocols do have substantial power. Among the most cited strengths are these:

• *Convergent consistency.* Properly designed gossip protocols, when not overwhelmed by a higher rate of incoming "events" than the information-carrying bandwidth of the underlying channels, should have a logarithmic mixing time – any new event will, with high probability, affect all nodes that need to learn about it within time logarithmic in the system size.

• *Emergent structure.* Earlier, we contrasted a classic deterministic protocol for building a spanning tree by leader-initiated flooding with a decentralized way of building such a tree using gossip. In the gossip style, the tree "emerges" from randomized pairwise interactions between peers. The term emergent structure is intended to evoke the image of a data structure that emerges with probability 1.0 in this manner. The structure may then continue to evolve over time as further gossip occurs.

• *Simplicity.* Most (but not all) gossip protocols are extremely simple and highly symmetric, with all participants running the same code.

• *Bounded load on participants.* Many classic (non-gossip) distributed protocols are criticized because they can generate high surge loads that overload individual components. Gossip is normally used in ways that produce strictly bounded worst-case loads on each component, eliminating the risk of disruptive load surges. In some situations, where network capacity is also a concern, peer-selection is further biased to control load imposed on network links.

• *Topology independence.* If running on a sufficiently connected networking substrate, and with sufficient bandwidth, a gossip protocol will often operate correctly on a great variety of underlying topologies.

• *Ease of local information discovery.* Many gossip protocols are used for purposes of discovery, for example to find a nearby resource (these are usually protocols in which gossip occurs between neighbors, not between arbitrarily distant peers). Unlike local flooding, which scales poorly, gossip would typically find local information less quickly but with bounded costs: perhaps, a constant or a delay logarithmic in the system size.

• *Robustness to transient network disruptions.* As time elapses, there are exponentially many routes by which information can flow from its source to its destinations. However, not all uses of gossip are robust in all ways. For example, unless data is self-verifying, dissemination protocols are often vulnerable to data corruption. Anti-entropy protocols may similarly be at risk if a replica becomes corrupted. And aggregation protocols are vulnerable not just to the introduction of faulty information, but also to computational errors that result in a faulty computation of the aggregate.

2.3 Appropriate roles for gossip

The foregoing discussion suggests a number of natural roles for gossip in large-scale event notification systems.

The earliest uses of gossip were to disseminate information in large-scale systems [22]. Scalability and robustness were cited as the primary benefits in these uses: the load on each node grows in a logarithmic manner as the system scales and information can be

reliability disseminated in the presence of a high proportion of node failures [20]. Such properties rely on the fact that each node samples network state randomly. This pseudo-randomness can nonetheless be controlled or "shaped". For example, Lpbcast [19] and Cyclon [15] are protocols in which each peer periodically selects another peer with which it gossips; they differ in the details of target selection, and in the way they merge information gathered through the gossip exchange with their own.

Generalizing these ideas, gossip may be used to create unstructured overlay networks, achieving properties close to those of random graphs [12]. Having used gossip to create such a graph, gossip protocols can also run over them, for example to create an overlay optimized with respect to an application-specific metric. For example, T-man builds overlays that use application-supplied quality functions to bias neighbor selection [10]. In [14], the gossip itself is biased; users with shared interests are structured into peer groups for file sharing, substantially improving response times in a search application.

Similarly, GosSkip [17] and Sub-2-Sub [13] build content-based publish-subscribe systems in which the overlay topology matches the subscription pattern. In GosSkip, subscriptions are organized into a skiplist structure so that events will be routed to interested subscribers in a logarithmic number of hops. In Sub-2-Sub, several gossip-protocols are layered to efficiently support range subscriptions. The lowest layer uses random peer sampling to ensure connectivity and robustness, a second layer creates clusters of "close" subscriptions, and the third layer structures overlapping subscriptions to ensure an exact and exhaustive dissemination of events.

This flexibility comes at a price. Gossip-based publish-subscribe overlays are often slow: the technology is wonderful for matching publishers with subscribers, but says little about getting events delivered rapidly, robustly, and with strong reliability properties. Indeed, we like to think of these kinds of applications as having two disjoint aspects: a gossip infrastructure that, in these cases, builds an overlay; and then a distinct dissemination structure that uses the overlay to reliably distribute events.

This way of thinking leads back to our current goals. We hope to systematically ask how gossip can be valuable in event-notification systems such as Quicksilver and in the applications that run over it. A number of options seem to be worth exploring. For example, as just seen, a gossip-constructed overlay network could be useful for efficient dissemination. In this case, Quicksilver itself would provide the "quality metrics" used to optimize the overlay, and the associated cost functions would reflect the mechanisms Quicksilver uses for dissemination and for recovery of lost packets.

More broadly, we hope to use gossip to materialize a form of distributed "picture" of the application network, which would become an input to an auto-configuration application that would generate configuration files. These would advise the end-user application (in addition to the Quicksilver event notification infrastructure) of the topology on which it should operate and the appropriate parameter settings to use. Later, as conditions evolve, the same approach could be used to reconfigure the running system so as to repair damage caused by a failure, or to integrate new components with the existing infrastructure.

Another possible role for gossip would be to track overall loads, loss rates and other status in the system. We have experience with a gossip-based system used for this purpose. Astrolabe is a distributed monitoring and data mining system that uses gossip to construct a virtual hierarchical database that can be queried much like a normal database [5]. The database is extremely useful for self-optimization and problem diagnosis. Because Astrolabe is fully replicated it has no single point of failure or load-related hot-spots, and the underlying gossip protocol remains robust even under stress that can shut down most other system functionality. In our new system, we believe aggregation mechanisms can play even more roles, including parameter setting and dynamic adaptation [11]. Aggregation can even be used for resource allocation, for example by using gossip to sort peers according to an application-specific metric [16].

Finally, we will use gossip to support background diffusion of system information that won't be needed immediately, but could be of high value "later". A tool permitting discovery of available information sources would be one possible use for such a mechanism. Other possibilities include mechanisms for tracking contact nodes or other services, finding information stored elsewhere in the network, etc. By using gossip to disseminate the underlying information, we can be certain that data will get through even if the system configuration changes (or is disrupted), and hence will be available when and where needed.

To exploit these kinds of gossip mechanisms, we need to tackle some significant software engineering issues that prior work has largely overlooked. To make gossip useful as a tool, one needs appropriate embeddings of these abstractions into the runtime environment. For these purposes, we propose to extend a feature of Cornell's Quicksilver platform, discussed below.

3. Quicksilver

Cornell's Quicksilver project [3][4] offers a scalable event notification infrastructure that can support strong properties on a per-topic basis. An application can subscribe to large numbers of communication channels, with the properties of each channel matched to the data it carries. Krzysztof Ostrowski is the lead architect and developer for Quicksilver, in collaboration with Ken Birman, Danny Dolev and Robbert van Renesse. We start

by reviewing prior work on Quicksilver, and then suggest some of the extensions our new effort will explore.

A key objective for Quicksilver is scalability in multiple dimensions: numbers of applications using the platform, numbers of event channels to which each application subscribes, data rates, tolerance of disruption, etc. Our underlying premise is that inadequate scalability has limited the uptake of group-multicast in general, and has prevented its widespread use in support of event notification. This sometimes manifests itself through throughput that degrades gracefully as the system is deployed into a larger setting, but more dramatic consequences are also observed. For example, many large-scale event notification platforms become unstable in large deployments, oscillating from very low throughput to overwhelmingly high data rates in which traffic generated by the platform can actually shut down the communications bus by swamping it with data, retransmissions, nack and ack messages and other forms of overhead – a so called broadcast storm effect. In designing Quicksilver, our goal was to demonstrate stability in this problematic domain.

This is not the right setting for a detailed discussion of the Quicksilver architecture. Instead, we summarize some key ideas very briefly:

- *Separation of concerns.* Quicksilver treats event dissemination separately from recovery of lost packets, flow control, and imp lementation of stronger consistency ("properties").
- *Regions of overlap.* A single node will often subscribe to many event channels. If each channel is treated as a separate multicast group, one encounters obvious problems of scale. Accordingly, Quicksilver maps from overlapping channels down to *regions*, defined to be sets of nodes with similar subscriptions. Dissemination is on a per-region basis; recovery is done in an aggregated manner over regions, etc.
- *Scalable recovery.* Quicksilver uses a novel hierarchy of token rings to achieve scalable detection of lost packets and, when possible, to recover data between peers in a region, offloading work from the sender.
- *Per-channel reliability properties.* The reliability properties of each channel can be matched to its role.
- *Managed runtime environment.* Quicksilver runs in managed settings, allowing it to leverage strong type checking, memory management, etc.

Details of the architecture and protocols appear in [3][4].

Quicksilver has been running since June 2006. For the moment, all our users are building datacenters – WAN scenarios are a goal once the new gossip-based mechanisms are available, but the current system doesn't run in WAN settings. In our datacenter experiments, we've set up groups with up to 200 nodes (larger runs are planned), than subjected them to extremely high throughputs and injected various forms of stress.

Up to the present, we have seen only minimal throughput degradation and no signs of instability or throughput fluctuations even in the largest configurations. In contrast, such problems are easy to provoke in most existing technologies for multicast in the same settings, even with much smaller groups of just 50 to 75 members [2]. Quicksilver can saturate a 100Mbit ethernet interconnect with just 20-40% CPU loads on the inexpensive PC's making up our test cluster; experiments with our prior systems peaked at about a tenth these data rates and generated much heavier loads. Perhaps most important, processes are able to access large numbers of groups. For this reason, when used to support event notification, Quicksilver can maintain steady performance even when each process joins as many as 8000 separate event channels [3][4]. Obviously, this capsule summary oversimplifies in some important ways (in particular, not all configurations of processes and event streams are supported), but they do give a sense of what the system should be able to achieve.

Of primary relevance here is the manner in which Quicksilver embeds event notification channels into Windows. Traditionally, event notification platforms have been treated as a free-standing technology that lives separately from the operating system. Quicksilver can be used this way too, through a conventional publish-subscribe infrastructure that generalizes the web services eventing standards (in [6] we discuss our reasons for extending these standards rather than working entirely within ws-notification or ws-eventing).

But Quicksilver also offers a second, deeper embedding into Windows in which event notification channels can be accessed either as a new kind of distributed *live object* visible in the file system side-by-side with other named objects. These objects are best understood as distributed abstract data types. A program accesses such an object much as it would access a file in Windows: given appropriate permissions, it can open the object, read the current state, and will receive events as the state is subsequently updated. This, however, is an illusion: the "object" is really an event channel, and the state is a checkpoint produced by some existing subscriber when a new program subscribes. State persistence is available, but optional.

We've emphasized the similarity between the way that a system such as Windows understands file "types" as an association between the data in some object and the programs that implement operations on that kind of object, and the way that Quicksilver associates a type with each event notification channel. For Quicksilver, the type corresponds to an object class, but also is associated with a definition of the properties the channel should implement. The effect is to confer a distributed semantics on the group of objects as a whole. The approach is flexible enough to support weak properties such as best-effort notification, stronger consensus-based properties

such as the virtual synchrony model, or even very strong models such as transactional 1-copy serializability. Quicksilver implements a domain-specific programming language within which the properties associated with each event channel can be specified. The system basically compiles these property definitions into pseudo-code which it can execute to achieve the desired behavior.

4. A unified platform

For our purposes, the key point of leverage involves the embedding of Quicksilver's live objects (event channels) into Windows. Consider the integration of abstract data types such as Excel spreadsheets or Word documents into the Windows file system. Windows uses the filename extension to understand the "type" of the object, allowing it to interpret operations on the object as method invocations on an appropriate application program. Web services standards are used in conjunction with these componentization mechanisms: active components such as the Excel application register their interfaces using the Web Services framework built into .net, at which point the Windows platform can function as a component integration environment using Web services standards and protocols to perform tasks such as method invocation. Of course, this component-to-component type system is somewhat primitive, but one could imagine taking the idea much further; indeed, there are projects underway at Microsoft to do just that. It isn't unreasonable to imagine that future versions of Windows will incorporate a full-fledged distributed type system at the component level.

As suggested above, Quicksilver extends Windows to support abstract data types with "live" content, and allows a variety of event stream providers to support the live aspects of the abstraction. A Quicksilver event notification channel has a name that can be visible in the file system name space, and a type, corresponding to the properties associated with the event channel. When an application binds itself to an event channel, Windows passes the binding event to Quicksilver, and we can perform type compatibility checking, or can even perform some kinds of dynamic type coercion (for example by introducing an encryption/decryption layer in order to integrate a component that doesn't support encryption with an event channel that requires stronger forms of security). The same mechanisms also work from the Windows shell: if a user right-clicks on a Quicksilver event channel, the shell extensions framework passes us the request. Quicksilver can then identify applications that can connect to this kind of channel, and can even generate dynamically created virtual folders, for example displaying thumbnail-size images from a video streaming application.

Quicksilver is thus on a path towards the same kind of tight integration with Quicksilver event streams as is seen with other Windows communications options such as DCOM. The approach enables developers to leverage existing Windows application development and debugging tools while benefiting from co-existence in a managed framework. If Windows evolves in the manner currently anticipated, type checking will become possible even across component boundaries. Because Quicksilver uses the CLR memory management layer, no copying occurs when a large object is multicast. Of course, such a positioning of the technology also brings challenges of its own (for example, to maximize performance in a managed environment requires protocol designs quite different from those one uses in a Linux/C multicast implementation [3]) but the problems are solvable and we believe the result is well worth the effort. We should comment that although Windows is our initial target, everything we are doing should port (using Mono) to Linux and would then be accessible from J2EE or even Corba applications.

This, then, is the core contribution of the present paper: a vision of how one might unify these three worlds: objects in a platform such as Windows on the one hand, and both gossip and of scalable event notification on the other, all in a single framework. A first step towards this vision requires that the Quicksilver multicast framework be separated from the mechanisms that embed Quicksilver objects into Windows; Ostrowski is already developing this capability as part of version 2.0 of the system. As is the case in Quicksilver today, the basic abstraction will be that of a distributed object having a "state" and an associated event stream. However, rather than assuming that the live content is transported by Quicksilver's reliable multicast protocols, there will be at least two possible communication infrastructures – the other being gossip-based. Down the road one might imagine additional options, such as an IP-TV streaming layer, or one focused on real-time communication.

Thus, referring back to the examples of gossip-based mechanisms mentioned in Section 3, one could build a gossip-based topology and configuration discovery service that, in effect, produces an annotated picture of the state of the system. An end-user could access that picture by clicking on an associated file name; doing so would launch some sort of browser capable of visualizing this kind of information and might let the user explore the network, for example to pin down a bottleneck that is impacting performance. Application programs could use the picture to configure themselves. And Quicksilver's event notification infrastructure could use that picture to construct overlays for disseminating events that use IP multicast when possible, but tunnel data through overlay trees where IP multicast is not feasible (and these same overlay networks would also be available to application designers, through some form of abstract data type). The remarkable robustness of the gossip protocols ensures that even when all else is disrupted, applications can still

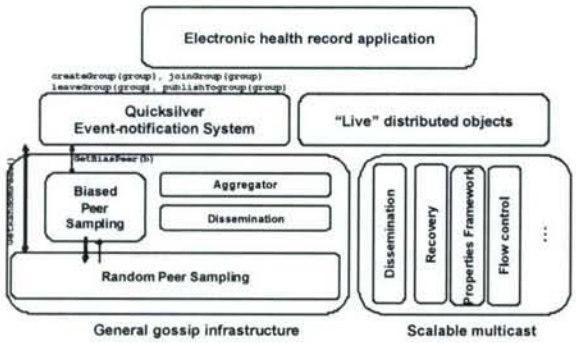


Figure 1: Overall System Architecture

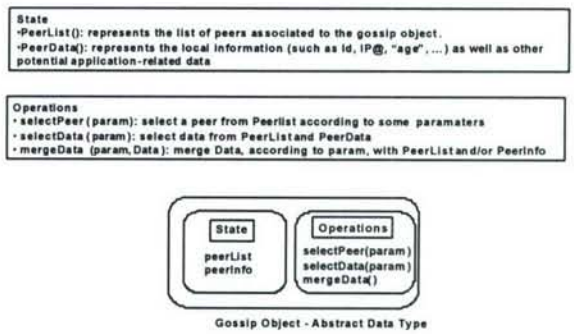


Figure 2: Generalized Implementation of a Gossip Object

monitor the system to set parameters, configure themselves, and adapt when conditions change.

But we believe we can do more than to simply import gossip functionality into Quicksilver. Gossip systems of the types we reviewed share substantial commonalities across their various presentations. For example, many gossip mechanisms require random peer selection, either within the full system (a kind of *anycast*) or within a set of neighbors of a node (a local variant on *anycast*). The thinking is that this and other low-level primitives can be standardized within the gossip subsystem, and then reused across gossip-based objects. Doing so poses interesting research challenges: if a single object employs *anycast*, one can implement a "greedy" solution. But suppose that on some single node there are tens or even hundreds of gossip-based objects, all using *anycast*. Could we aggregate, so that a single message can carry information on behalf of multiple objects?

One can pose similar questions at a higher level. Many gossip algorithms are highly stylized: the nature of a gossip exchange is rather similar across most gossip-based mechanisms, even if the details of what "state" is exchanged and how it is "merged" differ. This immediately suggests that one might design an abstract gossip state-machine that could be instantiated in multiple objects, parameterized with appropriate state marshalling and merge functions.

The resulting architecture is summarized in Figures 1 and 2. Figure 1 illustrates the overall system architecture, with the gossip infrastructure hosted side-by-side with the scalable multicast infrastructure and accessed either through a generalized publish-subscribe interface, or in the form of live distributed objects. As noted earlier, internal details for Quicksilver can be found in [3] and will not be repeated here. Figure 2 gives some additional detail for the gossip infrastructure.

5. Electronic health record example

We conclude the discussion by revisiting our electronic health record example, assuming now that the gossip mechanisms and the Quicksilver-based event notification solution are available side-by-side.

Let's start with roles for the gossip mechanisms. For the time being, we've decided to focus on uses in which the gossip components will be simple enough so that we can verify correctness, able to "sanity check" data collected from the environment, and unlikely to come under attack; these assumptions mitigate the security concerns mentioned earlier. For example, with gossip it isn't difficult to build a system that can track locations of system components: servers, client platforms, sensors, other devices. When a change occurs, the updated configuration should become visible with delay proportional to the log of the size of the system – in the scenarios we have in mind case, probably within 10 or 15 rounds of gossip. This capability could be the basis for a highly robust plug-and-play technology, whereby the health-care system would adapt in tens of seconds as conditions evolve. Although such a system might collect incorrect information about a platform that has some form of scrambled configuration state, the "damage" would be limited to the annotation of that component on the map, and the gossip objects can be designed to sense and reject implausible inputs.

Gossip could also be used to monitor system invariants (such as: "there should always be at least one instance of the auditing service"). Here, Quicksilver's notion of membership offers very rapid event detection and reaction, but if enough damage occurs while the system is running to seriously disrupt event notification, the gossip layer could guide a timely discovery of the problem and dynamic repair or adjustment of the parameters. The remarkable robustness of gossip mechanisms gives us reason for confidence that they will be able to continue to operate reliably even when other infrastructure components are severely degraded by a disruptive event.

Gossip can also be used to help system components connect themselves in appropriate ways. For example, a component might keep track of the locations of the various servers so that in the event of a fault that prevents

connection to one server, the clients using it can seamlessly roll over to others offering backup functionality. When the first server recovers, the clients can shift back. Gossip mechanisms can be used to monitor system health, assisting managers in diagnosing and repairing problems that arise because of software bugs or other disruptive events. If a firewall or server comes under attack (or just becomes overloaded), gossip based tracking mechanisms can help client systems discover the problem, identify fall-back options, and gracefully adapt.

Gossip also offers an antidote to certain kinds of fragility. For example, suppose that we want to track the physical location of patients in our hospital complex. In the most obvious standard implementation of an electronic health record system, one would probably place some sort of active component on the patient's gown or bed; it would continuously track its own location (somehow) and report that data to the central database. With gossip, new and potentially more robust options arise. Now, client systems can gossip with one-another about patient "sightings". With many observers and many paths by which information can spread, we obtain a patient location-tracking database at low cost, and with guarantees of extremely robust behavior even in the event of a disruptive condition, such as a malfunctioning application that generates extremely high network loads and loss rates. (Recall from our discussion of Astrolabe that a gossip management infrastructure might help in this case too, by assisting the system administrator in localizing the problem).

What about high-speed event notification and streaming? Our system could exploit this functionality in a great many ways. If we assume that health care records are, in effect, replicated throughout the system as a whole, when an update occurs, it will be important to consistently update all copies. Here we see a form of event notification that requires relatively strong reliability and delivery semantics – corresponding to a consensus-based model such as virtual synchrony or state machine replication, both available within Quicksilver as group "types". Event notification can support a publish-subscribe relationship between the database servers in the hospital and client systems operated in private practices and other satellite locations. Bedside or nursing station display systems may need to be refreshed. Similarly, if the update is relevant to a patient's prescriptions, the event might be pushed out to participating pharmacies. One can also imagine high-throughput event channels. For example, television cameras and other sensors monitoring infants in a neo-natal unit could stream images to the nursing station; pediatricians would be able to subscribe as necessary to keep an eye on their patients: a robust, scalable IP-TV architecture

The Quicksilver properties mechanisms would be beneficial here, by permitting the system to match the properties of each type of event channel, or live object, to the requirements associated with that category of object. In fact we doubt that there would be a huge number of cases, but there are clearly subsystems that would value real-time data delivery over other guarantees, subsystems that need the sorts of consistency afforded by virtual synchrony or state machine replication, and subsystems that need transactional "ACID" properties. These can all be supported, side-by-side, on a per-event-channel basis.

These example illustrates a point worth reiterating:b y using the publish-subscribe paradigm, the publishing side of the enterprise can be designed independently from the data consuming side; both can be incrementally extended over time as new applications are added, and will automatically accommodate varying runtime configurations. In effect, we are able to separate the information representation standards used within the system (including the hierarchy of topics) from the data sources and the data consumers. The communications infrastructure provides the needed guarantees, and when a new component is introduced, existing event-generating applications don't need to be modified. Because Quicksilver has a strong notion of types associated with event channels and live objects, we can do far more type checking than is traditionally feasible in publish-subscribe settings. For example, we can potentially ensure that the properties of a channel match the expectations of the application that binds itself to that channel. Moreover, to the extent that we need instant detection and reaction to a failure, because Quicksilver extends the publish-subscribe eventing model to also offer (optional) information about subscription changes when processes join and leave a channel, all sorts of rapid fault-tolerance mechanisms can be implemented.

We've avoided discussion of privacy and security issues, despite their central importance in electronic health care systems. This is in part because Quicksilver currently lacks a comprehensive security architecture, although we do have some ideas for how we might build one. Our thinking is to focus on capabilities enabled by the secure replication of security keys using the algorithms of Reiter [8][9] or Rodeh [7]; these offer ways to refresh keys when the set of nodes in the replication group (the event channel) changes because of a failure or a join. However, prior research has never explored scalability implications of these kinds of secure key replication schemes, and we believe the topic will require a substantial research effort to fully resolve. Use of security keys in gossip settings represents an additional intriguing option for study.

7. Research topics

Our vision raises a number of questions:

1. Given a proposed large-scale application, what is the most effective development methodology for mapping it down to application-specific functionality, as opposed to platform-supplied functionality? How should the developer make decisions concerning the aspects that are best matched to gossip communication, those best matched to event notification, and those that require hand-coded logic? Given that both gossip and event notification systems can support "guaranteed" properties, how should the developer decide which properties are needed by a given application, and how best to achieve them? Is there are large-scale methodology for specification of overall properties of a complex system that might lend itself to a formal verification process analogous to the ones used to reason about and ultimately prove correctness for non-distributed systems? Can the properties mechanisms used in Quicksilver today be extended to include gossip protocols?
2. If a single computer system supports multiple "live" data objects, high performance often requires that protocols be designed to amortize costs. Much of the innovation in Quicksilver is at this level: the system looks for ways to disseminate data, recover from packet loss and control data rates that are aggregated across potentially huge numbers of objects. When we introduce new classes of objects supported by gossip, the gossip infrastructure will need to address similar questions.
3. We alluded to the need to secure the platform, and to the risk that gossip mechanisms might be incapacitated by certain kinds of malicious behaviors. Our architecture poses significant opportunities for research on security, ranging from questions of precisely how one might secure a gossip protocol to broader issues of scalability that arise if an application subscribes to a large number of secured objects. How should one secure a high-speed event channel? What issues arise as one scales a security abstraction in a setting where each separate event channel or live object might have its own security requirements?
4. The creation of appropriate abstractions for the gossip infrastructure is an important challenge. At the lowest level, one imagines mechanisms for random peer selection, state exchange and merge, aggregation, etc. Ideally, these should be highly standardized. Yet some gossip protocols bias peer selection, implement "tricky" state exchange/merge mechanisms, or perform aggregation in unusual ways. Needed is a platform that can function well as a black box, and yet that can also expose functionality as needed.
5. We need to better understand the correct set of gossip mechanisms needed for purposes of self-management and self-configuration in Quicksilver. The modern internet is complex, and while it is easy to evoke a vision of an autonomic infrastructure that can support plug-and-play behavior in almost arbitrary settings, implementing that vision is quite a different matter.
6. Applications running on the event notification infrastructure will also need self-management and self-configuration functionality. Quicksilver's needs are somewhat peculiar to its role; will the same autonomic mechanisms that work for Quicksilver be adequate for other purposes, or are other kinds of gossip tools needed?
7. Obtaining high performance in large-scale settings that involve managed frameworks (C# in .net, in our case) is surprisingly hard [3]. It is likely that we will need to overcome similar challenges as we implement a gossip-based infrastructure and then tune it to cooperate cleanly with Quicksilver.
8. We commented that one key to scalability in Quicksilver is the mapping of event channels down to regions of approximate overlap – sets of nodes with similar subscription sets. A basic assumption underlying the system is that this can actually be done and that large systems will exhibit high degrees of overlap, or at least that they can be designed to have this property. But how can overlap regions be discovered in the first place? We are thinking that gossip mechanisms could be very useful in discovering applications and their "potential" subscription sets, enabling an offline analysis (perhaps with a human designer in the loop) to identify regions of overlap and configure Quicksilver. In contrast, the alternative of trying to discover regions at runtime by analysis of subscription patterns as programs come and go raises a number of thorny problems and may not be the best approach.

9. Conclusions

Scalable event notification systems capable of offering strong properties may be the key to enabling a new generation of trustworthy distributed applications, but only if they can be integrated naturally into the most powerful development environments and made autonomic: self-monitoring, self-configuring, and self-managing. For these latter purposes, we propose to build a new kind of distributed abstraction that embeds into Windows much like a typed object, but can be supported either by Quicksilver's scalable event architecture or by gossip-based protocols. A system realizing this vision is now under joint development at IRISA/INRIA in Rennes and at Cornell University.

7. References

[1] Reliable Distributed Systems Technologies, Web Services, and Applications. Birman, Kenneth P.

2005, XXXVI, 668 p. 145 illus., Hardcover ISBN: 0-387-21509-3

[2] A Review of Experiences with Reliable Multicast. K. P. Birman Software Practice and Experience Vol. 29, No. 9, pp, 741-774, July 1999

[3] Implementing Scalable Publish-Subscribe in a Managed Environment. Krzysztof Ostrowski, Ken Birman. In Submission (November, 2006).

[4] QuickSilver Scalable Multicast. Krzysztof Ostrowski, Ken Birman, and Amar Phanishayee. Cornell University Technical Report TR2006-2063 (April, 2006).

[5] Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. Robbert van Renesse, Kenneth Birman and Werner Vogels. ACM Transactions on Computer Systems, May 2003, Vol.21, No. 2, pp 164-206

[6] Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. Submitted to International Journal of Web Services Research.

[7] The Architecture and Performance of the Security Protocols in the Ensemble Group Communication System. Ohad Rodeh, Ken Birman, Danny Dolev. Journal of ACM Transactions on Information Systems and Security (TISSEC). Vol. 4, No 3, pp 289-319, Aug 2001

[8] A Security Architecture for Fault-Tolerant Systems. Michael K. Reiter, Kenneth P. Birman, Robbert van Renesse. ACM Trans. Comput. Syst. 12(4): 340-371 (1994)

[9] How to Securely Replicate Services. Michael K. Reiter, Kenneth P. Birman. ACM Trans. Program. Lang. Syst. 16(3): 986-1009 (1994)

[10] T-Man: Gossip-based overlay topology management. Mark Jelasity and Ozalp Babaoglu. In ESOA 2005, Revised Selected Papers, vol 3910 of LNCS, 1-15.

[11] Gossip-based aggregation in large dynamic networks. Mark Jelasity, Alberto Montresor and Ozalp Babaoglu. ACM Transactions on Computer Systems, 23(3): 219-252, August 2005.

[12] The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. Mark Jelasity, Rashid Guerraoui, Anne-Marie Kermarrec, Marteen van Steen. Middleware 2004, volume 3231 of LNCS, 79-98, Springer-Verlag, 2004.

[13] Sub-2-Sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. Spryros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec and Maarten van Steen. Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS), Santa-Barbara, CA, February 2006.

[14] Epidemic-style Management of Semantic Overlays for Content-based Searching. Spyros Voulgaris and Maarten van Steen, Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par), Lisbon, Portugal, August 2005.

[15] CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Spyros Voulgaris, Daniela Gavidia, Maarten van Steen. Journal of Network and Systems Management, vol. 13(2):197-217.

[16] Ordered Slicing of Very Large-Scale Overlay Networks. Mark Jelasity and Anne-Marie Kermarrec. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, 2006.

[17] GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists Principles. Rachid Guerraoui, Sidath Handurukande, Kevin Huguenin, Anne-Marie Kermarrec, Fabrice Le Fessant and Etienne Riviere. In The Sixth IEEE Conference on Peer to Peer Computing (P2P), Cambridge, UK, September, 2006.

[18] From Epidemics to Distributed Computing. Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. IEEE Computer, 37(5):60-67, May 2004.

[19] Lightweight Probabilistic Broadcast. Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov. ACM Transaction on Computer Systems, 21(4), November 2003.

[20] Probabilistic Reliable Dissemination in Large-Scale Systems. Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. IEEE Transactions on Parallel and Distributed Systems, 14(3), March 2003.

[21] BAR Gossip. Harry Li, Allen Clement, Edmund Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, Mike Dahlin, Proceedings of the 2006 USENIX Operating Systems Design and Implementation (OSDI), Nov 2006 .

[22] Epidemic algorithms for replicated database maintenance. Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry. ACM SIGOPS Operating Systems Review Volume 22 , Issue 1 (Jan., 1988), 8 - 32

Ricochet: Lateral Error Correction for Time-Critical Multicast

Mahesh Balakrishnan[†], Ken Birman[†], Amar Phanishayee[‡], Stefan Pleisch[†]
[†]*Cornell University and* [‡]*Carnegie Mellon University*
{*mahesh,ken,pleisch*}*@cs.cornell.edu, amarp+@cs.cmu.edu*

Abstract

Ricochet is a low-latency reliable multicast protocol designed for time-critical clustered applications. It uses IP Multicast to transmit data and recovers from packet loss in end-hosts using Lateral Error Correction (LEC), a novel repair mechanism in which XORs are exchanged between receivers and combined across overlapping groups. In datacenters and clusters, application needs frequently dictate large numbers of fine-grained overlapping multicast groups. Existing multicast reliability schemes scale poorly in such settings, providing latency of packet recovery that depends inversely on the data rate within a single group: the lower the data rate, the longer it takes to recover lost packets. LEC is insensitive to the rate of data in any one group and allows each node to split its bandwidth between hundreds to thousands of fine-grained multicast groups without sacrificing timely packet recovery. As a result, Ricochet provides developers with a scalable, reliable and fast multicast primitive to layer under high-level abstractions such as publish-subscribe, group communication and replicated service/object infrastructures. We evaluate Ricochet on a 64-node cluster with up to 1024 groups per node: under various loss rates, it recovers almost all packets using LEC in tens of milliseconds and the remainder with reactive traffic within 200 milliseconds.

1 Introduction

Clusters and datacenters play an increasingly important role in the contemporary computing spectrum, providing back-end computing and storage for a wide range of applications. The modern datacenter is typically composed of hundreds to thousands of inexpensive commodity blade-servers, networked via fast, dedicated interconnects. The software stack running on a single blade-server is a brew of off-the-shelf software: commercial operating systems, proprietary middleware, managed run-time environments and virtual machines, all standardized to reduce complexity and mitigate maintenance costs.

The last decade has seen the migration of time-critical applications to commodity clusters. Application domains ranging from computational finance to air-traffic control and military communication have been driven by scalability and cost concerns to abandon traditional real-time environments for COTS datacenters. In the process, they give up conservative - and arguably unnecessary - guarantees of real-time performance for the promise of massive scalability and multiple nines of timely availability, all at a fraction of the running cost. Delivering on this promise within expanding and increasingly complex datacenters is a non-trivial task, and a wealth of commercial technology has emerged to support clustered applications.

At the heart of commercial datacenter software is *reliable multicast* — used by publish-subscribe and data distribution layers [5, 7] to spread data through clusters at high speeds, by clustered application servers [1, 4, 3] to communicate state, updates and heartbeats between server instances, and by distributed caching infrastructures [2, 6] to rapidly update cached data. The multicast technology used in contemporary industrial products is derivative of protocols developed by academic researchers over the last two decades, aimed at scaling metrics like throughput or latency across dimensions as varied as group size [10, 17], numbers of senders [9], node and network heterogeneity [12], or geographical and routing distance [18, 21]. However, these protocols were primarily designed to extend the reach of multicast to massive networks; they are not optimized for the failure modes of datacenters and may be unstable, inefficient and ineffective when retrofitted to clustered settings. Crucially, they are not designed to cope with the unique scalability demands of time-critical fault-tolerant applications.

We posit that a vital dimension of scalability for clustered applications is the *number of groups* in the system. All the uses of multicast mentioned above induce large numbers of overlapping groups. For example, a computational finance calculator that uses a topic-based pub-sub system to subscribe to a fraction of the equities on the stock market will end up belonging in many multicast groups. Multiple such applications within a datacenter - each subscribing to different sets of equities - can result in arbitrary patterns of group overlap. Similarly, data caching or replication at fine granularity can result in a single node hosting many data items. Replication driven by high-level objectives such as locality, load-balancing or fault-tolerance can lead to distinct overlapping replica sets - and hence, multicast groups - for each item.

In this paper, we propose Ricochet, a time-critical re-

liable multicast protocol designed to perform well in the multicast patterns induced by clustered applications. Ricochet uses IP Multicast [15] to transmit data and recovers lost packets using *Lateral Error Correction* (LEC), a novel error correction mechanism in which XOR repair packets are probabilistically exchanged between receivers and combined across overlapping multicast groups. The latency of loss recovery in LEC depends inversely on the aggregate rate of data in the system, rather than the rate in any one group. It performs equally well in any arbitrary configuration and cardinality of group overlap, allowing Ricochet to scale to massive numbers of groups while retaining the best characteristics of state-of-the-art multicast technology: even distribution of responsibility among receivers, insensitivity to group size, stable proactive overhead and graceful degradation of performance in the face of increasing loss rates.

1.1 Contributions

- We argue that a critical dimension of scalability for multicast in clustered settings is the number of groups in the system.
- We show that existing reliable multicast protocols have recovery latency characteristics that are inversely dependent on the data rate in a group, and do not perform well when each node is in many low-rate multicast groups.
- We propose Lateral Error Correction, a new reliability mechanism that allows packet recovery latency to be independent of per-group data rate by intelligently combining the repair traffic of multiple groups. We describe the design and implementation of Ricochet, a reliable multicast protocol that uses LEC to achieve massive scalability in the number of groups in the system.
- We extensively evaluate the Ricochet implementation on a 64-node cluster, showing that it performs well with different loss rates, tolerates bursty loss patterns, and is relatively insensitive to grouping patterns and overlaps - providing recovery characteristics that degrade gracefully with the number of groups in the system, as well as other conventional dimensions of scalability.

2 System Model

We consider patterns of multicast usage where each node is in many different groups of small to medium size (10 to 50 nodes). Following the IP Multicast model, a group is defined as a set of receivers for multicast data, and senders do not have to belong to the group to send to it. We expect each node to receive data from a large set of distinct senders, across all the groups it belongs to.

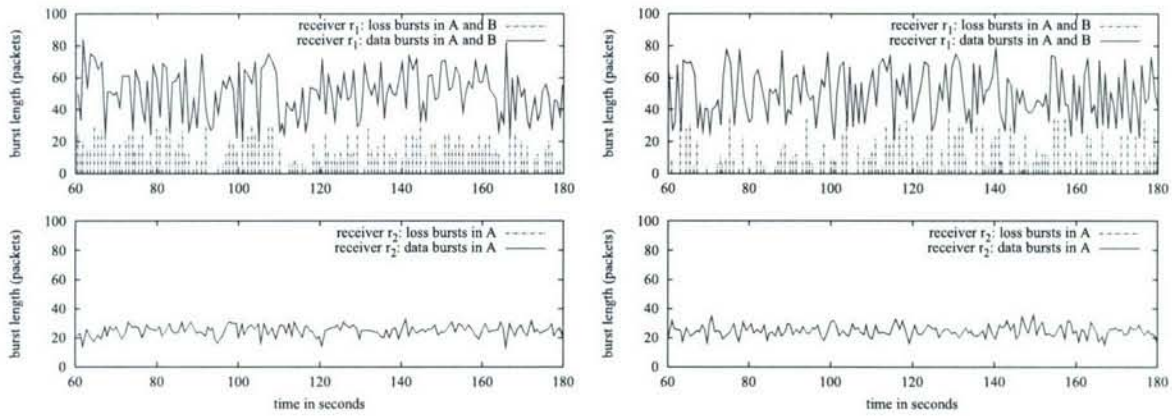
Where does Loss occur in a Datacenter? Datacenter networks have flat routing structures with no more than two or three hops on any end-to-end path. They are typically over-provisioned and of high quality, and packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are inexpensive and easily overloaded; even with high-capacity network interfaces, the commodity OS often drops packets due to buffer overflows caused by traffic spikes or high-priority threads occupying the CPU. Hence, our loss model is one of short packet bursts dropped at the end-host receivers at varying loss rates.

Figure 1 strongly indicates that loss in a datacenter is (a) bursty and (b) independent across end-hosts. In this experiment, a receiver r_1 joins two multicast groups A and B, and another receiver r_2 in the same switching segment joins only group A. From a sender located multiple switches away on the network, we send per-second data bursts of around 25 1KB packets to group A and simultaneously send a burst of 0-50 packets to group B, and measure packet loss at both receivers. We ran this experiment on two networks: a 64-node cluster at Cornell with 1.3 Ghz receivers and the Emulab testbed at Utah with 2 Ghz receivers, all nodes running Linux 2.6.12.

The top graphs in Figure 1 show the traffic bursts and loss bursts at receiver r_1, and the bottom graphs show the same information for r_2. We can see that r_1 gets overloaded and drops packets in bursts of size 1-30 packets, whereas r_2 does not drop any packets — importantly, around 30% of the packets dropped by r_1 are in group A, which is common to both receivers. Hence, loss is both bursty and independent across nodes. Together, these graphs indicate strongly that loss occurs due to buffer overflows at receiver r_1.

The example in Figure 1 is simplistic - each incoming burst of traffic arrives at the receiver within a small number of milliseconds - but conveys a powerful message: it is very easy to trigger significant bursty loss at datacenter end-hosts. The receivers in these experiments were running empty and draining packets continuously out of the kernel, with zero contention for the CPU or the network, whereas the settings of interest to us involve time-critical, possibly CPU-intensive applications running on top of the communication stack.

Further, we expect multi-group settings to intrinsically exhibit bursty incoming traffic of the kind emulated in this experiment — each node in the system receives data from multiple senders in multiple groups and it is likely that the inter-arrival time of data packets at a node will vary widely, even if the traffic rate at one sender or group is steady. In some cases, burstiness of traffic could also occur due to time-critical application behavior - for example, imagine an update in the value of a stock quote triggering off activity in several system components, which then multicast information to a replicated central data-



(a) Cornell 64-node Cluster

(b) Utah Emulab Testbed

Figure 1: Datacenter Loss is bursty and uncorrelated across nodes: receiver r_1 (top) joins groups A and B and exhibits bursty loss, whereas receiver r_2 (bottom) joins only group A and experiences zero loss.

store. If we assume that each time-critical component processes the update within a few hundred microseconds, and that inter-node socket-to-socket latency is around fifty microseconds (an actual number from our experimental cluster), the central datastore could easily see a sub-millisecond burst of traffic. In this case, the componentized structure of the application resulted in bursty traffic; in other scenarios, the application domain could be intrinsically prone to bursty input. For example, a financial calculator tracking a set of hundred equities with correlated movements might expect to receive a burst of a hundred packets in multiple groups almost instantaneously.

3 The Design of a Time-Critical Multicast Primitive

In recent years, multicast research has focused almost exclusively on application-level routing mechanisms, or overlay networks ([13] is one example), designed to operate in the wide-area without any existing router support. The need for overlay multicast stems from the lack of IP Multicast coverage in the modern internet, which in turn reflects concerns of administration complexity, scalability, and the risk of multicast 'storms' caused by misbehaving nodes. However, the homogeneity and comparatively limited size of datacenter networks pose few scalability and administration challenges to IP Multicast, making it a viable and attractive option in such settings. In this paper, we restrict ourselves to a more traditional definition of 'reliable multicast', as a reliability layer over IP Multicast. Given that the selection of datacenter hardware is typically influenced by commercial constraints, we believe that any viable solution for this context must be able to run on any mix of existing commodity routers and OS software; hence, we focus exclusively on mechanisms that act at the application-level, ruling out schemes which require router modification, such as PGM [19].

3.1 The Timeliness of (Scalable) Reliable Multicast Protocols

Reliable multicast protocols typically consist of three logical phases: *transmission* of the packet, *discovery* of packet loss, and *recovery* from it. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols, the time taken to discover packet loss dominates recovery latency heavily in the kind of settings we are interested in. The key insight is that *the discovery latency of reliable multicast protocols is usually inversely dependent on data rate*: for existing protocols, the rate of outgoing data at a single sender in a single group. Existing schemes for reliability in multicast can be roughly divided into the following categories:

ACK/timeout: RMTP [21], RMTP-II [22]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [21]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger time-outs at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

Gossip-Based: Bimodal Multicast [10], lpbcast [17]. Receivers periodically gossip histories of received packets

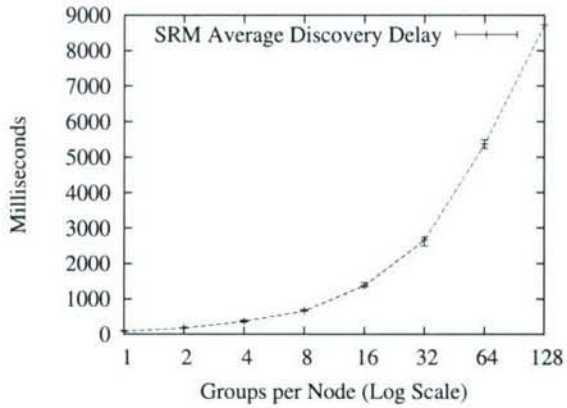


Figure 2: SRM's Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs.

with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery latency is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [10]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact.

NAK/Sender-based Sequencing: SRM [18]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery latency is thus proportional to the inter-send time at any single sender to a single group - a receiver can't discover a loss in a group until it receives the next packet from the same sender to that group - and consequently depends on the sender's data transmission rate to the group. To illustrate this point, we measured the performance of SRM as we increased the number of groups each node belonged in, keeping the throughput in the system constant by reducing the data rate within each group - as Figure 2 shows, discovery latency of lost packets degrades linearly as each node's bandwidth is increasingly fragmented and each group's rate goes down, increasing the time between two consecutive sends by a sender to the same group. Once discovery occurs in SRM, lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

Sender-based FEC [20, 23]: Forward Error Correction schemes involve multicasting redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating c repair packets for every r data packets, such that any r of the combined set of $r + c$ data and repair packets is sufficient to recover the original r data packets; we term this (r, c) parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out r data packets to that group. Fast, efficient encodings such as Tornado codes [11] make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, software distribution or internet radio.

Receiver-based FEC [9]: Of the above schemes, ACK-based protocols are intrinsically unsuited for time-critical multi-sender settings, while sender-based sequencing and FEC limit discovery latency to inter-send time at a single sender within a single group. Ideally, we would like discovery latency to be independent of inter-send time, and combine the scalability of a gossip-based scheme with the tunability of FEC. Receiver-based FEC, first introduced in the Slingshot protocol [9], provides such a combination: receivers generate FEC packets from incoming data and exchange these with other randomly chosen receivers. Since FEC packets are generated from *incoming* data at a receiver, the timeliness of loss recovery depends on the rate of data multicast in *the entire group*, rather than the rate at any given sender, allowing scalability in the number of senders to the group.

Slingshot is aimed at single-group settings, recovering from packet loss in time proportional to that group's data rate. Our goal with Ricochet is to achieve recovery latency dependent on the rate of data incoming at a node *across all groups*. Essentially, we want recovery of packets to occur as quickly in a thousand 10 Kbps groups as in a single 10 Mbps group, allowing applications to divide node bandwidth among thousands of multicast groups while maintaining time-critical packet recovery. To achieve this, we introduce Lateral Error Correction, a new form of receiver-generated FEC that probabilistically combines receiver-generated repair traffic across multiple groups to drive down packet recovery latencies.

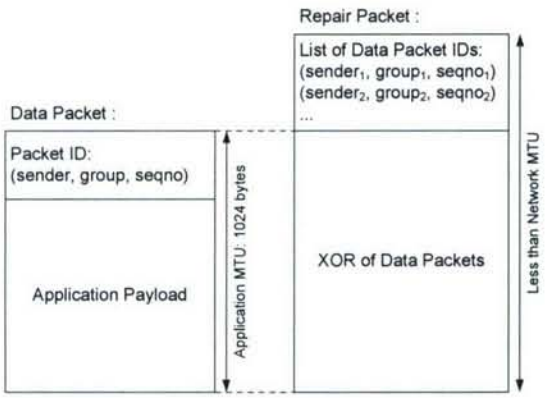


Figure 3: Ricochet Packet Structure

4 Lateral Error Correction and the Ricochet protocol

In Ricochet, each node belongs to a number of groups, and receives data multicast within any of them. The basic operation of the protocol involves generating XORs from incoming data and exchanging them with other randomly selected nodes. Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 3 shows the structure of these two packet types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from r data packets allows recovery of one of them, if all the other $r-1$ data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets.

At the core of Ricochet is the LEC engine running at each node that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind LEC is the notion that repair packets sent by a node to another node can be composed from data in any of the multicast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur within time that's inversely proportional to the aggregate rate of data in all these groups. Figure 4 illustrates this idea: n_1 has groups A and B in common with n_2, and hence it can generate and dispatch repair packets that contain data from both these groups. n_1 needs to wait only until it receives 5 data packets in either A *or* B before it sends a repair packet, allowing faster recovery of lost packets at n_2.

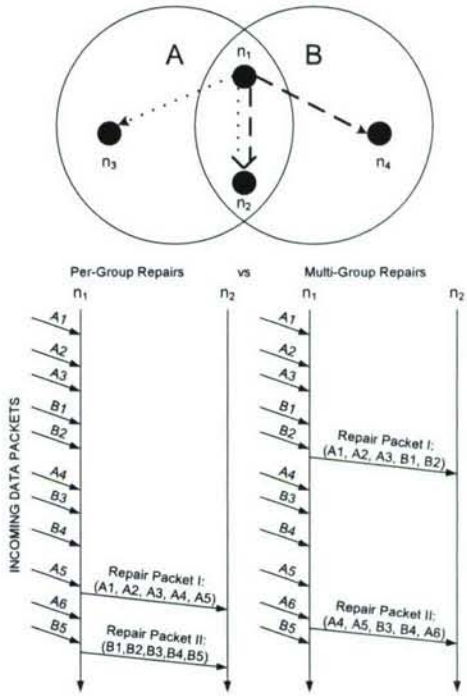


Figure 4: LEC in 2 Groups: Receiver n_1 can send repairs to n_2 that combine data from both groups A and B.

While combining data from different groups in outgoing repair packets drives down recovery time, it tampers with the coherent tunability that single group receiver-based FEC provides. The *rate-of-fire* parameter in receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every r data packets, c repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for LEC is to combine repair traffic for multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. Coherent, Tunable Per-Group Overhead: For every data packet that a node receives in a group with rate-of-fire (r, c), it sends out *an average of* c *repair packets* including that data packet to other nodes in the group.
2. Randomness: Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

LEC supports overlapping groups with the same r component and different c values in their rate-of-fire parameter. In LEC, the rate-of-fire parameter is translated into the following guarantee: For every data packet d that a node receives in a group with rate-of-fire (r, c), it selects an average of c nodes from the group randomly and sends each

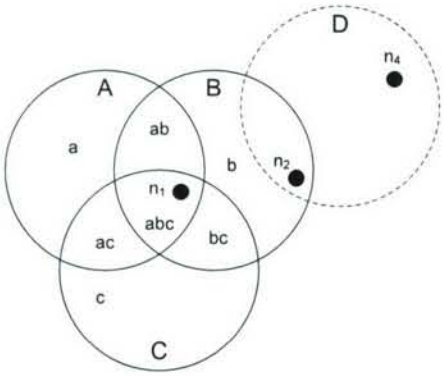


Figure 5: n_1 belongs to groups A, B, C: it divides them into disjoint regions abc, ab, ac, bc, a, b, c

of these nodes exactly one repair packet that includes d. In other words, the node sends an average of c repair packets containing d to the group. In the following section, we describe the algorithm that LEC uses to compose and dispatch repair packets while maintaining this guarantee.

4.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same LEC algorithm and supporting code runs at every node - and hence, we can describe its operation from the vantage point of a single node, n_1.

4.1.1 Regions

The LEC engine running at n_1 divides n_1's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that n_1 belongs to. Figure 5 shows the regions in a hypothetical system, where n_1 is in three groups, A, B and C. We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e, abc is a region formed by the intersection of A, B and C. In our example, the neighborhood set of n_1 is carved into seven regions: abc, ac, ab, bc, a, b and c, essentially the power set of the set of groups involved. Readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case; we are only concerned with non-empty intersections, the cardinality of which is bounded by the number of nodes in the system, as each node belongs to exactly one intersection (see Section 4.1.4). Note that n_1 does not belong to group D and is oblivious to it; it observes n_2 as belonging to region b, rather than bd, and is not aware of n_4's existence.

4.1.2 Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, LEC selects targets randomly from *each region*. The number of targets selected from a region is set such that:

1. It is proportional to the size of the region
2. The total number of targets selected, across regions, is equal to the c value of the group

Hence, for a given group A with rate-of-fire (r, c_A), the number of targets selected by LEC in a particular region, say abc, is equal to $c_A * \frac{|abc|}{|A|}$, where $|x|$ is the number of nodes in the region or group x. We denote the number of targets selected by LEC in region abc for packets in group A as c_A^{abc}. Figure 6 shows n_1 selecting targets for repairs from the regions of A.

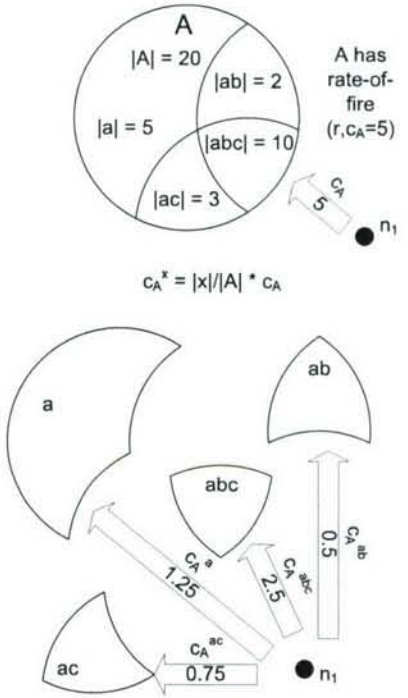


Figure 6: n_1 selects proportionally sized chunks of c_A from the regions of A

Note that LEC may pick a different number of targets from a region for packets in a different group; for example, c_A^{abc} differs from c_B^{abc}. Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size c has to be selected from the group, selecting targets from regions corresponds to *stratified sampling* [14], a technique from statistical theory.

4.1.3 Why select targets from regions?

Selecting targets from regions instead of groups allows LEC to construct repair packets from multiple groups; since we know that all nodes in region ab are interested in data from groups A and B, we can create composite

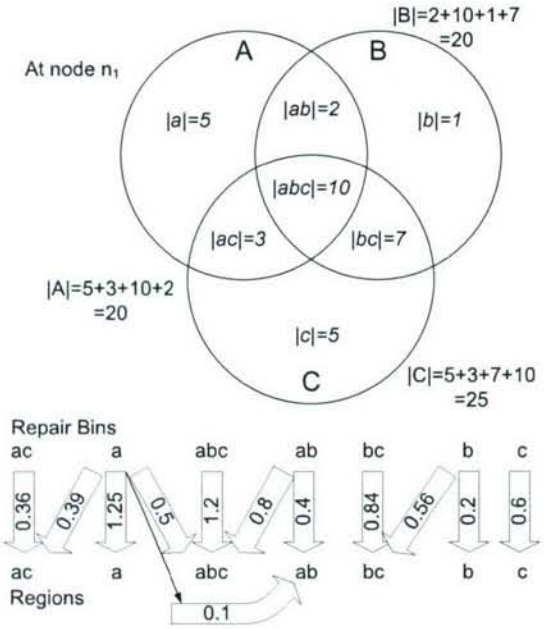


Figure 7: Mappings between repair bins and regions: the repair bin for ab selects 0.4 targets from region ab and 0.8 from abc for every repair packet. Here, $c_A = 5$, $c_B = 4$, and $c_C = 3$.

repair packets from incoming data packets in both groups and send them to nodes in that region.

Single-group receiver-based FEC [9] is implemented using a simple construct called a *repair bin*, which collects incoming data within the group. When a repair bin reaches a threshold size of r, a repair packet is generated from its contents and sent to c randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems simple; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size r, it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups A, B, and C, how many targets should we select from region abc for this repair packet - c_A^{abc}, c_B^{abc}, or c_C^{abc}? One possible solution is to pick the maximum of these values. If $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$, then we would select c_A^{abc}. However, a data packet in group B, when added to the repair bin for the region abc would be sent to an average of c_A^{abc} targets in the region; resulting in more repair packets containing that data packet sent to the region than required (c_B^{abc}), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group B than required by its (r, c_B) value; a similar argument holds for data packets in group C as well.

Algorithm 1 Algorithm for Setting Up Repair Bins

1: **Code at node n_i:**
2: **upon** Change in Group Membership **do**
3: **while** L not empty {*L is the list of regions*} **do**
4: Select and remove the region $R_i = abc...z$ from L with highest number of groups involved (break ties in any order)
5: Set $R_t = R_i$
6: **while** $R_t \neq \epsilon$ **do**
7: set c_{min} to $min(c_A^{R_t}, c_B^{R_t} ...)$, where {A,B,...} is the set of groups forming R_t
8: Set number of targets selected by R_i's repair bin from region R_t to c_{min}
9: Remove G from R_t, for all groups G where $c_G^{R_t} = c_{min}$
10: For each remaining group G' in R_t, set $c_{G'}^{R_t} = c_{G'}^{R_t} - c_{min}$

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups A and B than required, resulting in a lower fraction of packets recovered from LEC. To rectify this we send the additional compensating repair packets to the region abc from the repair bins for regions a and b. The repair bin for region a would select $c_A^{abc} - c_C^{abc}$ destinations, on an average, for every repair packet it generates; this is in addition to the c_A^a destinations it selects from region a.

A more sophisticated version of the above strategy involves iteratively obtaining the required repair packets from regions involving the remaining groups; for instance, we would have the repair bin for ab select the minimum of c_A^{abc} and c_B^{abc} - which happens to be c_B^{abc} - from abc, and then have the repair bin for a select the remainder value, $c_A^{abc} - c_B^{abc}$, from abc. Algorithm 1 illustrates the final approach adopted by LEC, and Figure 7 shows the output of this algorithm for an example scenario. A repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

4.1.4 Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that n_1 is part of. The algorithm has complexity $O(I \cdot d)$, where I is the number of populated regions (i.e, with one or more nodes in them), and d is the maximum number of groups that form a region. Note that I at n_1 is bounded from above by the cardinality of the set of nodes that share a multicast

group with n_1, since regions are disjoint and each node exists in exactly one of them. d is bounded by the number of groups that n_1 belongs to.

4.2 Implementation Details

Our implementation of Ricochet is in Java. Below, we discuss the details of the implementation, along with the performance optimizations involved - some obvious and others subtle.

4.2.1 Repair Bins

A Ricochet repair bin is a lightweight structure holding an XOR and a list of data packets, and supporting an add operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in Figure 7, the repair bin for abc has to select 1.2 targets from abc, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

4.2.2 Staggering for Bursty Loss

A crucial algorithmic optimization in Ricochet is *staggering* - also known as interleaving [23] - which provides resilience to bursty loss. Given a sequence of data packets to encode, a stagger of 2 would entail constructing one repair packet from the 1st, 3rd, 5th... packets, and another repair packet from the 2nd, 4th, 6th... packets. The stagger value defines the number of repairs simultaneously being constructed, as well as the distance in the sequence between two data packets included in the same repair packet. Consequently, a stagger of i allows us to tolerate a loss burst of size i while resulting in a proportional slowdown in recovery latency, since we now have to wait for $O(i*r)$ data packets before despatching repair packets.

In conventional sender-based FEC, staggering is not a very attractive option, providing tolerance to very small bursts at the cost of multiplying the already prohibitive loss discovery latency. However, LEC recovers packets so quickly that we can tolerate a slowdown of a factor of ten without leaving the tens of milliseconds range; additionally, a small stagger at the sender allows us to tolerate very large bursts of lost packets at the receiver, especially since the burst is dissipated among multiple groups and senders. Ricochet implements a stagger of i by the simple expedient of duplicating each logical repair bin into i instances; when a data packet is added to the logical repair bin, it is actually added to a particular instance of the repair bin, chosen in round-robin fashion. Instances of a duplicated repair bin behave exactly as single repair bins do, generating repair packets and sending them to regions when they get filled up.

4.2.3 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [24]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the LEC algorithm. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at n_1 - and consequently the group and intersection sizes - does not include n_1 itself.

4.2.4 Membership and Failure Detection

Ricochet can plug into any existing membership and failure detection infrastructure, as long as it is provided with reasonably up-to-date views of per-group membership by some external service. In our implementation, we use simple versions of Group Membership (GMS) and Failure Detection (FD) services, which execute on high-end server machines. If the GMS receives a notification from the FD that a node has failed, or it receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s). The GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node n_{55} joins group A, the update sent by the GMS to every node in A would be a 3-tuple: *(Join, A, n_{55})*. Individual nodes process these updates to construct multi-group views relative to their own membership.

Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this can be easily done by partitioning the service on group id and running each partition on a different server. For instance, one machine is responsible for groups A and B, another for C and D, and so on. Similarly, the FD can be partitioned on a topological criterion; one machine on each rack is responsible for monitoring other nodes on the rack by pinging them periodically. For fault-tolerance, each partition of the GMS can be replicated on multiple machines using a strongly consistent protocol like Paxos. The FD can have a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain. We believe that

such a semi-centralized solution is appropriate and sufficient in a datacenter setting, where connectivity and membership are typically stable. Crucially, the protocol itself does not need consistent membership, and degrades gracefully with the degree of inconsistency in the views; if a failed node is included in a view, performance will dip fractionally in all the groups it belongs to as the repairs sent to it by other nodes are wasted.

4.2.5 Performance

Since Ricochet creates LEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to: Instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to c (from the rate-of-fire) in the worst case - which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

4.2.6 Buffering and Loss Control

LEC - like any other form of FEC - works best when losses are not in concentrated bursts. Ricochet maintains an application-level buffer with the aim of minimizing in-kernel losses, serviced by a separate thread that continuously drains packets from the kernel. If memory at end-hosts is constrained and the application-level buffer is bounded, we use customized packet-drop policies to handle overflows: a randomly selected packet from the buffer is dropped and the new packet is accommodated instead. In practice, this results in a sequence of almost random losses from the buffer, which are easy to recover using FEC traffic. Whether the application-level buffer is bounded or not, it ensures that packet losses in the kernel are reduced to short bursts that occur only during periods of overload or CPU contention. We evaluate Ricochet against loss bursts of up to 100 packets, though in practice we expect the kind of loss pattern shown in 1, where few bursts are greater than 20-30 packets, even with highly concentrated traffic spikes.

4.2.7 NAK Layer for 100% Recovery

Ricochet recovers a high percentage of lost packets via the proactive LEC traffic; for certain applications, this probabilistic guarantee of packet recovery is sufficient and even desirable in cases where data 'expires' and there is no utility in recovering it after a certain number of milliseconds. However, the majority of applications require 100% recovery of lost data, and Ricochet uses a reactive NAK layer to provide this guarantee. If a receiver does not recover a packet through LEC traffic within a timeout period after discovery of loss, it sends an explicit NAK to the sender and requests a retransmission. While this NAK layer does result in extra reactive repair traffic, two factors separate it from traditional NAK mechanisms: firstly, recovery can potentially occur very quickly - within a few hundred milliseconds - since for almost all lost packets discovery of loss takes place within milliseconds through LEC traffic. Secondly, the NAK layer is meant solely as a backup mechanism for LEC and responsible for recovering a very small percentage of total loss, and hence the extra overhead is minimal.

4.2.8 Optimizations

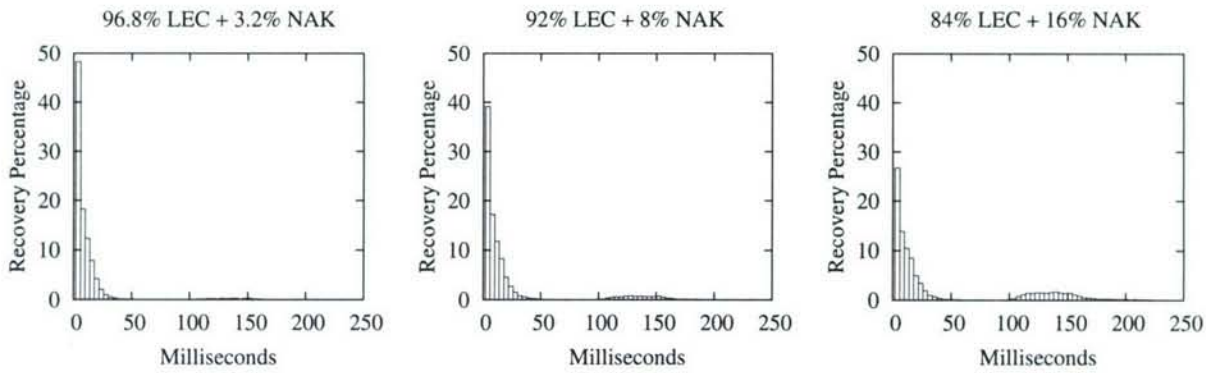
Ricochet maintains a buffer of unusable repair packets that enable it to utilize incoming repair packets better. If one repair packet is missing exactly one more data packet than another repair packet, and both are missing at least one data packet, Ricochet obtains the extra data packet by XORing the two repair packets. Also, it maintains a list of unusable repair packets which is checked intermittently to see if recent data packet recoveries and receives have made any old repair packets usable.

4.2.9 Message Ordering

As presented, Ricochet provides multicast reliability but does not deliver messages in the same order at all receivers. We are primarily concerned with building an extremely rapid multicast primitive that can be used by applications that require unordered reliable delivery as well as layered under ordering protocols with stronger delivery properties. For instance, Ricochet can be used as a reliable transport by any of the existing mechanisms for total ordering [16] — in separate work [8], we describe one such technique that predicts out-of-order delivery in datacenters to optimize ordering delays.

5 Evaluation

We evaluated our Java implementation of Ricochet on a 64-node cluster, comprising of four racks of 16 nodes each, interconnected via two levels of switches. Each node has a single 1.3 GHz CPU with 512 Mb RAM, runs Linux 2.6.12 and has two 100 Mbps network interfaces, one for control and the other for experimental traffic. Typical socket-to-socket latency within the cluster is around 50 microseconds. In the following experiments, for a given loss rate L, three different loss models are used:
· **uniform** - also known as the Bernoulli model [25] - refers to dropping packets with uniform probability equal to the loss rate L.



(a) 10% Loss Rate **(b) 15% Loss Rate** **(c) 20% Loss Rate**

Figure 8: Distribution of Recoveries: LEC + NAK for varying degrees of loss

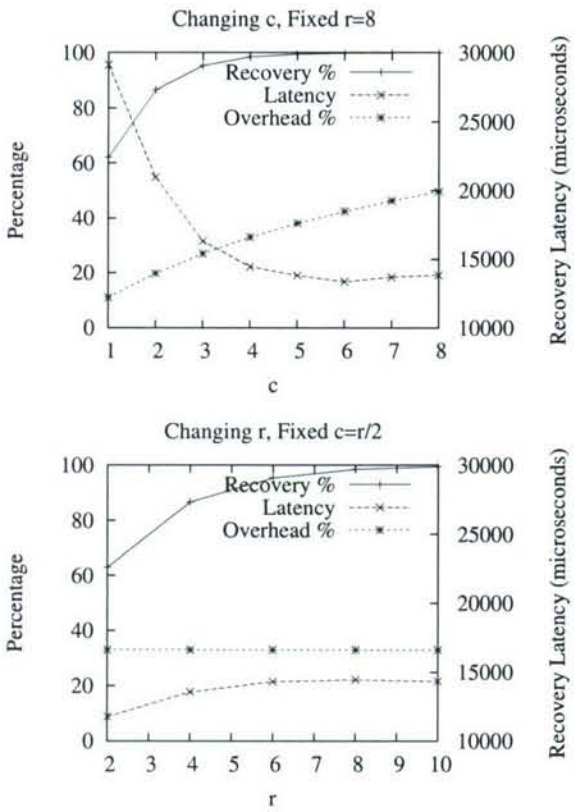


Figure 9: Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire (r, c).

· **bursty** involves dropping packets in equal bursts of length b. The probability of starting a loss burst is set so that each burst is of exactly b packets and the loss rate is maintained at L. This is not a realistic model but allows us to precisely measure performance relative to specific burst lengths.

· **markov** drops packets using a simple 2-state markov chain, where each node alternates between a lossy and a lossless state, and the probabilities are set so that the average length of a loss burst is m and the loss rate is L, as described in [25].

In experiments with multiple groups, nodes are assigned to groups at random, and the following formula is used to relate the variables in the grouping pattern: $n * d = g * s$, where n is the number of nodes in the system (64 in most of the experiments), d is the degree of membership, i.e. the number of groups each node joins, g is the total number of groups in the system, and s is the average size of each group. For example, in a 16-node setting where each node joins 512 groups and each group is of size 8, g is set to $\frac{16*512}{8} \approx 1024$. Each node is then assigned to 512 randomly picked groups out of 1024. Hence, the grouping patterns for each experiment is completely represented by a (n, d, s) tuple.

For every run, we set the sending rate at a node such that the total system rate of incoming messages is 64000 packets per second, or 1000 packets per node per second. Data packets are 1K bytes in size. Each point in the following graphs - other than Figure 8, which shows distributions for single runs - is an average of 5 runs. A run lasts 30 seconds and produces $\approx$ 2 million receive events in the system.

5.1 Distribution of Recoveries in Ricochet

First, we provide a snapshot of what typical packet recovery timelines look like in Ricochet. Earlier, we made

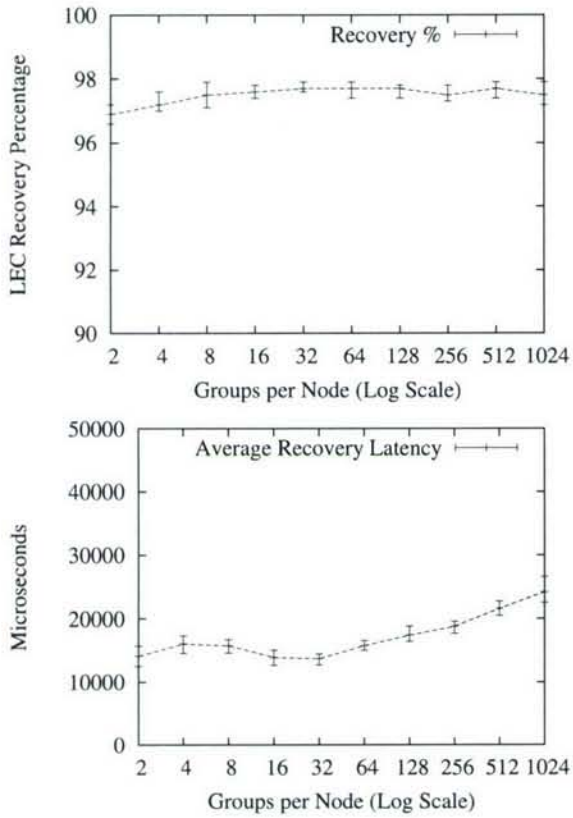


Figure 10: Scalability in Groups

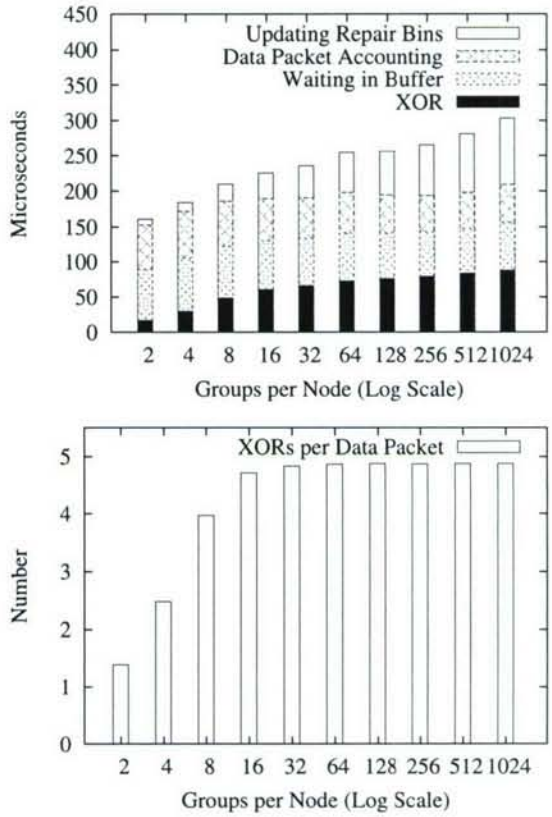


Figure 11: CPU time and XORs per data packet

the assertion that Ricochet discovers the loss of almost all packets very quickly through LEC traffic, recovers a majority of these instantly and recovers the remainder using an optional NAK layer. In Figure 8, we show the histogram of packet recovery latencies for a 16-node run with degree of membership $d = 128$ and group size $s = 10$. We use a simplistic NAK layer that starts unicasting NAKs to the original sender of the multicast 100 milliseconds after discovery of loss, and retries at 50 millisecond intervals. Figure 8 shows three scenarios: under uniform loss rates of 10%, 15%, and 20%, different fractions of packet loss are recovered through LEC and the remainder via reactive NAKs. These graphs illustrate the meaning of the LEC recovery percentage: if this number is high, more packets are recovered very quickly without extra traffic in the initial segment of the graphs, and less reactive overhead is induced by the NAK layer. Importantly, even with a recovery percentage as low as 84% in Figure 8(c), we are able to recover all packets within 250 milliseconds with a crude NAK layer due to early LEC-based discovery of loss. For the remaining experiments, we will switch the NAK layer off and focus solely on LEC performance; also, since we found this distribution of recovery latencies to be fairly representative, we present only the percentage of lost packets recovered using LEC and the average latency of these recoveries. Experiment Setup: $(n = 16, d = 128, s = 10)$, Loss Model: Uniform, [10%, 15%, 20%].

5.2 Tunability of LEC in multiple groups

The Slingshot protocol [9] illustrated the tunability of receiver-generated FEC for a single group; we include a similar graph for Ricochet in Figure 9, showing that the rate-of-fire parameter (r, c) provides a knob to tune LEC's recovery characteristics. In Figure 9.a, we can see that increasing the c value for constant $r = 8$ increases the recovery percentage and lowers recovery latency by expending more overhead - measured as the percentage of repair packets to all packets. In Figure 9.b, we see the impact of increasing r, keeping the ratio of c to r - and consequently, the overhead - constant. For the rest of the experiments, we set the rate-of-fire at $(r = 8, c = 5)$. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Uniform, 1%.

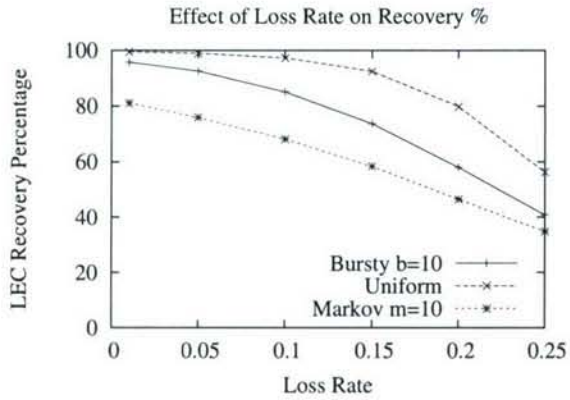


Figure 12: Impact of Loss Rate on LEC

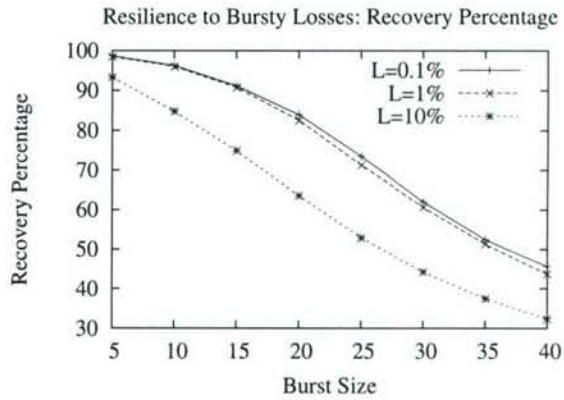


Figure 13: Resilience to Burstiness

5.3 Scalability

Next, we examine the scalability of Ricochet to large numbers of groups. Figure 10 shows that increasing the degree of membership for each node from 2 to 1024 has almost no effect on the percentage of packets recovered via LEC, and causes a slow increase in average recovery latency. The x-axis in these graphs is log-scale, and hence a straight line increase is actually logarithmic with respect to the number of groups and represents excellent scalability. The increase in recovery latency towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section.

For a comparison point, we refer readers back to SRM's discovery latency in Figure 2: in 128 groups, SRM discovery took place at 9 seconds. In our experiments, SRM recovery took place roughly 4 seconds after discovery in all cases. While fine-tuning the SRM implementation for clustered settings should eliminate that 4 second gap between discovery and recovery, at 128 groups Ricochet surpasses SRM's best possible recovery performance of 5 seconds by between 2 and 3 orders of magnitude.

Though Ricochet's recovery characteristics scale well in the number of groups, it is important that the computational overhead imposed by the protocol on nodes stays manageable, given that time-critical applications are expected to run over it. Figure 11 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a log x-axis shows that per-packet processing time increases logarithmically with the number of groups - doubling the number of groups results in a constant increase in processing time. The increase in processing time towards the latter half of the graph is due to the increase in the number of repair bins with the number of groups. While we considered 1024 groups adequate scalability, Ricochet can potentially scale to more groups with further optimization, such as creating bins only for occupied regions. In the current implementation, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second. Figure 11 also shows the average number of XORs per incoming data packet. As stated in section 4.2.2, the number of XORs stays under 5 - the value of c from the rate-of-fire (r, c). Experiment Setup: $(n = 64, d = *, s = 10)$, Loss Model: Uniform, 1%.

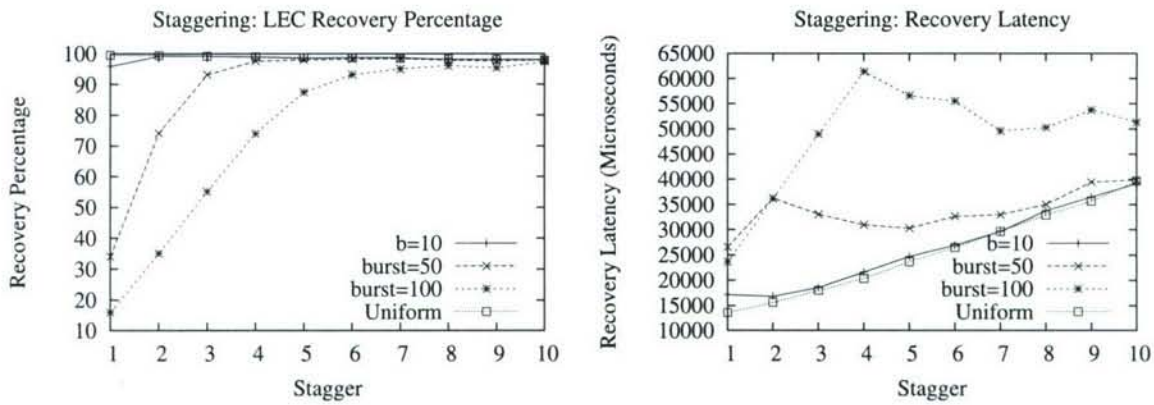


Figure 14: Staggering allows Ricochet to recover from long bursts of loss.

5.4 Loss Rate and LEC Effectiveness

Figure 12 shows the impact of the Loss Rate on LEC recovery characteristics, under the three loss models. Both LEC recovery percentages and latencies degrade gracefully: with an unrealistically high loss rate of 25%, Ricochet still recovers 40% of lost packets at an average of 60 milliseconds. For uniform and bursty loss models, recovery percentage stays above 90% with a 5% loss rate; markov does not fare as well, even at 1% loss rate, primarily because it induces bursts much longer than its average of 10 - the max burst in this setting averages at 50 packets. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: *.

5.5 Resilience to Bursty Losses

As we noted before, a major criticism of FEC schemes is their fragility in the face of bursty packet loss. Figure 13 shows that Ricochet is naturally resilient to small loss bursts, without the stagger optimization - however, as the burst size increases, the percentage of packets recovered using LEC degrades substantially. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty.

However, switching on the stagger optimization described in Section 4.2.2 increases Ricochet's resilience to burstiness tremendously, without impacting recovery latency much. Figure 14 shows that setting an appropriate stagger value allows Ricochet to handle large bursts of loss: for a burst size as large as 100, a stagger of 6 enables recovery of more than 90% lost packets at an average latency of around 50 milliseconds. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty, 1%.

5.6 Effect of Group and System Size

What happens to LEC performance when the average group size in the cluster is large compared to the total number of nodes? Figure 15 shows that recovery percentages are almost unaffected, staying above 99% in this scenario, but recovery latency is impacted by more than a factor of 2 as we triple group size from 16 to 48 in a 64-node setting. Note that this measures the impact of the size of the group relative to the entire system; receiver-based FEC has been shown to scale well in a single isolated group to hundreds of nodes [9]. Experiment Setup: $(n = 64, d = 128, s = *)$, Loss Model: Uniform, 1%.

While we could not evaluate to system sizes beyond 64 nodes, Ricochet should be oblivious to the size of the entire system, since each node is only concerned with the groups it belongs to. We ran 4 instances of Ricochet on each node to obtain an emulated 256 node system with each instance in 128 groups, and the resulting recovery percentage of 98% - albeit with a degraded average recovery latency of nearly 200 milliseconds due to network and CPU contention - confirmed our intuition of the protocol's fundamental insensitivity to system size.

6 Future Work

One avenue of research involves embedding more complex error codes such as Tornado [11] in LEC; however, the use of XOR has significant implications for the design of the algorithm, and using a different encoding might require significant changes. LEC uses XOR for its simplicity and speed, and as our evaluation showed, we obtain properties on par with more sophisticated encodings, including tunability and burst resilience. We plan on replacing our simplistic NAK layer with a version optimized for bulk transfer, providing an efficient backup for LEC when sustained bursts occur of hundreds of packets or more. Another line of work involves making the parameters for LEC - such as rate-of-fire and stagger - adaptive, reacting to meet varying load and network characteristics. We are currently working with industry partners to layer Ricochet under data distribution, publish-subscribe and web-service interfaces, as well as building protocols with stronger ordering and atomicity properties over it.

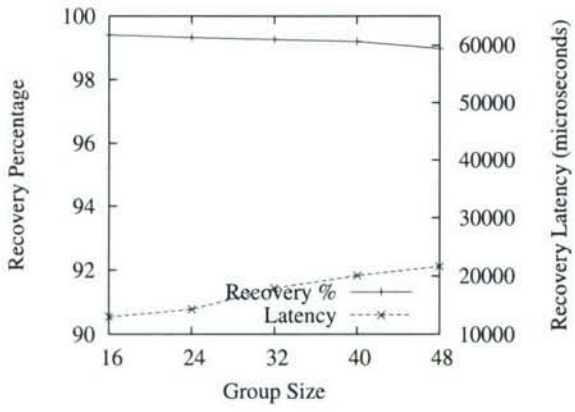


Figure 15: Effect of Group Size

7 Conclusion

We believe that the next generation of time-critical applications will execute on commodity clusters, using the techniques of massive redundancy, fault-tolerance and scalable communication currently available to distributed systems practitioners. Such applications will require a multicast primitive that delivers data at the speed of hardware multicast in failure-free operation and recovers from packet loss within milliseconds irrespective of the pattern of usage. Ricochet provides applications with massive scalability in multiple dimensions - crucially, it scales in the number of groups in the system, performing well under arbitrary grouping patterns and overlaps. A clustered communication primitive with good timing properties can ultimately be of use to applications in diverse domains not normally considered time-critical - e-tailers, online webservers and enterprise applications, to name a few.

Acknowledgments

We received invaluable comments from Dave Andersen, Danny Dolev, Tudor Marian, Art Munson, Robbert van Renesse, Emin Gun Sirer, Niraj Tolia and Einar Vollset. We would like to thank our shepherd Mike Dahlin, as well as all the anonymous reviewers of the paper.

References

[1] Bea weblogic. `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic`, 2006.

[2] Gemstone gemfire. `http://www.gemstone.com/products/gemfire/enterprise.php`, 2006.

[3] Ibm websphere. `www.ibm.com/software/webservers/appserv/was/`, 2006.

[4] Jboss. `http://labs.jboss.com/portal/`, 2006.

[5] Real-time innovations data distribution service. `http://www.rti.com/products/data_distribution/index.html`, 2006.

[6] Tangosol coherence. `http://www.tangosol.com/html/coherence-overview.shtml`, 2006.

[7] Tibco rendezvous. `http://www.tibco.com/software/messaging/rendezvous.jsp`, 2006.

[8] M. Balakrishnan, K. Birman, and A. Phanishayee. Plato: Predictive latency-aware total ordering. In *IEEE SRDS*, 2006.

[9] M. Balakrishnan, S. Pleisch, and K. Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*, 2005.

[10] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98 Conference*, pages 56–67, New York, NY, USA, 1998. ACM Press.

[12] Y. Chawathe, S. McCanne, and E. A. Brewer. Rmx: Reliable multicast for heterogeneous networks. In *INFOCOM*, pages 795–804, 2000.

[13] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *ACM SIGCOMM*, pages 55–67, New York, NY, USA, 2001. ACM Press.

[14] W. G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.

[15] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[17] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[18] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[19] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The pgm reliable multicast protocol. *IEEE Network*, 17(1):16–22, Jan 2003.

[20] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[21] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[22] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, Apr. 1998. IETF Internet Draft.

[23] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.

[24] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[25] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *INFOCOM*, pages 345–352, 1999.

MISTRAL: Efficient Flooding in Mobile Ad-hoc Networks*

Stefan Pleisch† Mahesh Balakrishnan‡ Ken Birman‡ Robbert van Renesse‡

†Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland

‡Department of Computer Science
Cornell University, Ithaca, NY 14853, USA

stefan.pleisch@epfl.ch {mahesh|ken|rvr}@cs.cornell.edu

ABSTRACT

Flooding is an important communication primitive in mobile ad-hoc networks and also serves as a building block for more complex protocols such as routing protocols. In this paper, we propose a novel approach to flooding, which relies on proactive compensation packets periodically broadcast by every node. The compensation packets are constructed from dropped data packets, based on techniques borrowed from forward error correction. Since our approach does not rely on proactive neighbor discovery and network overlays it is resilient to mobility.

We evaluate the implementation of Mistral through simulation and compare its performance and overhead to purely probabilistic flooding. Our results show that Mistral achieves a significantly higher node coverage with comparable overhead.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*;
C.4 [**Computer-Communication Networks**]: Performance of Systems—*Reliability, availability, and serviceability*;
C.4 [**Computer-Communication Networks**]: Performance of Systems—*Fault tolerance*

General Terms

Algorithms, reliability

Keywords

Mobile ad hoc networks, MANET, flooding, forward error correction, compensation

*Our effort is supported by the Swiss National Science Foundation (SNF), NSF Trust STC, the NSP NetNOSS program, and the DARPA ACERT program.

MobiHoc'06, May 22–25, 2006, Florence, Italy.

1. INTRODUCTION

Mobile ad hoc networks (MANETs) have received much attention in recent years. A MANET is a multi-hop wireless network without fixed infrastructure, in which nodes can be mobile. MANETs are increasingly important because wireless communication is rapidly becoming ubiquitous. Potential applications range from military and disaster response applications to more traditional urban problems such as finding desired products or services in a city. The devices themselves are diverse, including PDAs, cell phones, sensors, laptops, etc. Many new protocols have been proposed to solve the technical problems confronted in MANETs and to offer platform support for applications that collect and exploit the data available in such settings.

Because of the lack of a fixed communication infrastructure, flooding in MANETs [10] is an important communication primitive and also serves as a building block for more complex protocols such as AODV [21] or ODMRP [16]. *Flooding* is the mechanism by which a node, receiving flooded message m for the first time, rebroadcasts m once. We distinguish between flooding and *broadcast*, which is a transmission that is received by all nodes within transmission range of the broadcasting node. Flooding usually covers all the nodes in a network, but can also be limitted to a set of nodes that is defined by a geographical area (also called *geocast flooding* [14]) or by the time-to-live (TTL) parameter of m. Thus, a node receiving the flooded message only rebroadcasts it if it is within the specified area or if the message's TTL is greater than 0.

Unfortunately, flooding has been shown to be susceptible to contention even in reasonably dense networks [18]. Indeed, flooding leads to a large amount of redundant messages that consume scarce resources such as bandwidth and power and cause contention, collisions and thus additional packet loss. Every node receives the message from every neighbor within transmission range, except when messages are lost due to contention and collisions. This problem is known as the *broadcast storm* problem [18]. Because flooding is important in MANET applications, there is a clear need for storm-resistant flooding protocols that operate efficiently. However, reducing the number of redundant broadcasts leads to a lower degree of reliability. Hence, the challenge we face is to strike a balance between message overhead (i.e., the level of redundancy) and reliability.

To reduce the number of redundant messages, two ba-

sic classes of mechanisms have been proposed: (1) imposing a (partial) routing overlay structure; and (2) selectively dropping messages. Approaches in (1) build and maintain a (partial) routing overlay structure in the ad hoc network, which is used to efficiently broadcast the flooded message. For instance, only nodes that are part of a multicast tree rebroadcast the message [20]. Other approaches in this category are [3, 8, 17]. With mobile nodes the underlying routing structure needs to be frequently changed, incurring high maintenance costs and generally reduced reliability during the restructuring. In contrast, approaches in (2) do not rely on an explicit underlying routing structure. Instead, each node uses local information to make an independent decision whether to rebroadcast or to drop the flooded message. The simplest approach in this class is purely probabilistic flooding [18], in which messages are rebroadcast with a certain fixed probability. While probabilistic flooding reduces the number of broadcasts, when applied naively it simply recreates our earlier problem: poorly connected nodes (those with few neighbors) may fail to receive a flooded message. This consideration has motivated a number of more complex approaches, such as the algorithms given in [28, 24].

In our paper, we focus on class (2) but propose a new mechanism to reduce the number of missed flooded messages. We start with purely probabilistic flooding [18] but compensate for dropped data packets by periodically broadcasting *compensation packets*. Every compensation packet encodes a set of packets that have been dropped (i.e., that are not rebroadcast) by the sender. A node's neighbors, upon receipt of such a packet, can recover missing packets if it already has received and buffered a sufficient percentage of the packets that were used in constructing the compensation packet.

Even when a node has lost too many packets to reconstruct missing data, the compensation packets provide information that can be used to identify the loss. We include a secondary recovery mechanism that kicks in when a node discovers an unrecoverable loss, and part of our task in the evaluation presented here is to quantify the tradeoff between the additional message overhead versus increased reliability.

We have implemented Mistral and simulate its performance on JiST/SWANS, a simulation package that lets the developer run code in an emulated environment. Our results show that compensation packets significantly increase coverage when compared to probabilistic flooding with comparable overhead.

The remainder of the paper is structured as follows: Section 2 overviews the problem of flooding and places our work in the context of earlier work. In Section 3 we introduce the Mistral algorithm. Section 4 provides a simple analysis of Mistral. In Section 5, we present the simluation results and measure Mistral's performance. We conclude the paper with Section 6.

2. FLOODING IN MANETS

In any flooding mechanism, one must balance reliability against message overhead. On the one hand, increasing reliability generally involves sending a greater number of redundant messages and thus incurs a higher message overhead. In this worst case, the system risks provoking broadcast storms. Yet redundant messages are needed to reach all nodes and to recover from packet loss, hence reducing the overhead will generally decrease reliability.

The broadcast storm problem is so common in flooding algorithms that it has engendered a whole area of research. Storm-sensitive flooding approaches can be broadly classified into two classes: *local-knowledge-based* and *overlay-based*. Local-knowledge-based approaches decide on whether to rebroadcast or drop a flooded message solely on the basis of local information. Most commonly, they use information from received broadcasts to adaptively determine the forwarding policy. Such algorithms are a natural fit for MANETs, as they do not need to maintain any kind of complex node-to-node state that might need to be adapted in the event of mobility or other topology changes. In contrast, overlay-based approaches structure the node field according to some (local) topology, and then use topological information to efficiently implement flooding and reliability. The problem here is that if nodes have low quality connections to neighbors and/or are in motion, the overlay structure must be adapted. As a consequence, a high rate of management messages may be required, and if a flooded message is propagated while the overlay is out of date, that message may experience a high loss rate. In the worst case, the system might end up in a state of churn, constantly adapting the overlay but never managing to achieve the high quality of flooding that the overlay is intended to support.

We now briefly overview existing work and assign it to the corresponding class. For reasons of brevity, our review is deliberately partial; we focus on results that inspired our work here, or that have been widely cited in the literature. For a more comprehensive overview that includes a comparison of some of the major flooding approaches the reader is referred to [26].

2.1 Overlay-Based Approaches

As just indicated, we use the term *overlay* very broadly. For us, an overlay-based approaches is an algorithm that superimposes a routing structure onto the ad hoc network in support of flooding and rebroadcast. Depending on the position of a node in this overlay, it decides to either rebroadcast a flooded packet, or to only process and then drop it. While overlays provide a convenient mechanism to reduce the message overhead of flooding and to increase reliability, they suffer from the need to reconfigure the overlay when connectivity changes or if the nodes are mobile. Restructuring adds overhead but also increases the likelihood that messages will be lost, and thus may decrease coverage of the flooding protocol.

Ni *et al.* [18] propose to structure the nodes into clusters. Their solution rebroadcasts a packet in a manner that depends on the node's position in the cluster: only cluster head and gateway nodes rebroadcast.

In [8], the goal is to provide low-latency flooding. This is in part achieved by minimizing the collisions and interference. Gandi *et al.* show that an optimal solution to this problem is NP complete, instead, they propose an approximation algorithm. They construct a multicast tree and compute a rebroadcasting schedule such that the expected rate of collisions will be low.

Other approaches are based on the approximation of (minimal) connected dominating sets (MCDS), e.g., [5] [3]. Informally, a dominating set (DS) contains a subset of all nodes such that every node not in the DS is adjacent to one in the DS. Thus, a DS creates a virtual backbone that can be used to efficiently flood messages. It has been shown that

the creation of an MCDS is NP-complete. Thus, most approaches attempt to find a sufficiently good approximation to a MCDS.

A number of approaches rely on two-hop neighbor information to select nodes that rebroadcast the message. These approaches require that *hello* messages containing neighbor information are exchanged between the nodes.

For instance, in the Double-Covered Broadcast (DCB) [17], node n collects information about the two-hop neighbor set. Among its one-hop neighbors it then picks nodes that rebroadcast the message (called *forward node*) such that (1) the rebroadcast by the forward node covers the two-hop neighbors, and (2) the one-hop neighbors that are no forward nodes are within range of at least two rebroadcasts by forward nodes. The reception of the message by the forward node is implicitly acknowledged when n overhears the rebroadcast.

The scalable broadcast algorithm (SBA) [20] also uses two-hop neighbor knowledge, but employs a different approach to select the forward nodes.

With node mobility, the two-hop neighbor sets need to be updated frequently. Otherwise, the neighbor sets become outdated and reliability drops (as observed in [17]).

2.2 Local-Knowledge-Based Approaches

Local-knowledge-based approaches generally decide on a per-node basis whether to rebroadcast a particular flooded message. In the simplest case, each node flips a coin and rebroadcasts messages with a certain probability p [18]. We call this approach *purely probabilistic flooding (PPF)*.

There are a number of variants on this basic idea. For example, one set of algorithms base the rebroadcast decision either on the number of already overheard rebroadcasts, or on the distance or location of the overheard rebroadcast's sender [18]. The idea underlying these schemes is that the additional coverage gained by rebroadcasting decreases with the number of overheard rebroadcasts and decreasing distance to neighboring rebroadcasting nodes. However, it takes time to collect these statistics, delaying the rebroadcast decision, hence a potentially high latency is introduced to every flooded message. In [25], Tseng *et al.* extend earlier approaches in [18] to allow nodes to dynamically adapt threshold values such as the rebroadcast counter.

In [28] Zhang and Agrawal propose an approach that is a combination of the counter-based and probabilistic method of [18]. Instead of using a static rebroadcast probability p, they adjust p according to the information collected by the counters. While this makes p adaptable, it becomes dependent upon other fixed parameters that need to be carefully selected (e.g., timeouts).

Dynamic Gossip [24] relies on local density awareness to adjust the rebroadcast probability p of the one-hop neighbors. Its correctness and suitability relies on the assumption that the nodes are uniformly distributed. Density information is collected using a relay-ping method.

In [15], Kowalski and Pelc propose a broadcasting algorithm with optimal lower bounds in their model. They consider only stationary nodes and ajust the broadcast probability accordingly.

Haas *et al.* [9] study what they term a *phase transition phenomena*. This work shows that purely probabilistic flooding (called *gossiping* in [9]) in an ad hoc network has a *bimodal* delivery distribution. Their simulations reveal that either almost every node receives the message, or virtually none. To reduce the likelihood of the latter case, they explore a variety of approaches, such as adapting the rebroadcast probability to the density or the distance to the flooding source. Sasson *et al.* [23] theoretically explore the same phenomena based on percolation theory and conclude that there exists a threshold $\bar{p} < 1$ such that for any $p > \bar{p}$ the node coverage is close to 1, while for $p < \bar{p}$ the coverage is very low. Hence, increasing p much beyond $\bar{p}$ is not very useful.

Any approach that bases rebroadcast decision on observation of neighbors and on overheard broadcasts is at risk of using stale information if nodes might move before the information is used. MANETs, of course, can have a high degree of mobility, hence neither of these approaches is ideal.

Mistral's compensation mechanisms is orthogonal to these approaches. Indeed, were we building a production deployment of flooding in a real-world setting, we would be inclined to combine Mistral with one of these others (as should be clear, the ideal choice of underlying mechanism depends upon the anticipated density of nodes and level of mobility; no single solution stands out as uniformly superior to the others). By using such a hybrid scheme, we could parameterize the underlying solution to keep overheads low, accepting a modest risk that flooded packets would fail to reach some nodes. Compensation packets could then be used to overcome this low level of residual losses.

3. MISTRAL

Traditional flooding suffers from the problem of redundant message reception, once per neighbor. Even in a reasonably connected network, the same message is received multiple times by every node, which is inefficient, wastes valuable resources, and can create contention in the transmission medium.

Selective rebroadcasting of flooded messages is a way to limit the number of redundant transmissions. Instead of simply rebroadcasting the message a node evaluates a local function $\mathcal{F}$ and then uses the outcome of this computation to decide whether to forward the message. In its simplest form, this function returns its result based on some static probability (corresponding to PPF). More complex functions take into account additional topological (e.g., the number of neighbors) or statistical information (e.g., the number of overheard rebroadcasts). The downside of selective flooding is that a flooding may no longer reach all intended nodes. In particular, if a node has only few neighbors, none of these neighbors may rebroadcast the message. Selective flooding thus balances message overhead against reliability.

Mistral finds some middle ground by introducing a new mechanism that allows us to fine-tune the balance between message overhead and reliability. The key idea is to extend selective flooding approaches by compensating for messages that are not rebroadcast. This compensation is based on a technique borrowed from forward error correction (FEC). Every incoming data packet (dp) is either rebroadcast or added to a compensation packet (cp). The compensation packet is broadcast at regular intervals and allows the receivers to recover one missing data packet.

3.1 Forward Error Correction

In its simplest form, Forward Error Correction (FEC) [11,

19, 22] creates l repair packets for every m data packets such that any m out of the resulting $(m+l)$ packets is enough to recover the original m data packets [11]. Traditional applications of FEC generate l repair packets for every m data packets and inject them into a data stream, which insulates the receiver from at most l packet losses. One of the fundamental advantages of FEC is that it imposes a constant overhead on the system and has easily understandable behavior under arbitrary network conditions. However, this simple form of FEC was developed for streaming settings, where a single sender is transmitting data at a high, steady rate such as in bulk file transfers [6] or in a video or audio feeds [7]. Part of our challenge is to develop a FEC solution matched to the characteristics of a MANET.

3.2 Algorithm

We noted earlier that Mistral can be built on top of any local-knowledge-based flooding approach. In the current implementation of the system, we use purely probabilistic flooding, mostly because this approach is extremely simple and is intuitively easy to visualize. Recall that in PPF, a node rebroadcasts a flooded message with static probability p. Although PPF might not be an ideal choice of algorithm in a practical deployment, the algorithm has no "hidden" effects that might make it hard to interpret our experimental findings.

Upon reception of a data packet, every node evaluates the function $\mathcal{F} : dp \longmapsto \{true|false\}$. In its most basic form, $\mathcal{F}$ takes a data packet as input and returns a boolean. If it returns true, dp is rebroadcast; otherwise, dp is added to the current compensation packet. When the number of data packets contained in a compensation packet passes a certain threshold c, the compensation packet is broadcast. We call c the *compensation rate*. Thus, a compensation packet is broadcast for every c data packets that are not rebroadcast.

Algorithm 1 presents the algorithm in more detail: Procedure `process` delivers the data packet to the application and decides whether to rebroadcast the packet or add it to the compensation packet; `composeCompensationPac` builds the compensation packet; and `runRecovery` attempts to recover data packets from stored compensation packets when a new data packet is delivered to the application. Finally, procedure `expand` is used for level-2 recovery, which is presented in Section 3.2.2. The secondary recovery mechanism discussed in the introduction is not included in Algorithm 1.

3.2.1 Composition of a Compensation Packet

In this section, we assume that data packets are of fixed size, e.g., 512 bytes, and contain the payload, a sender ID and some locally unique sequence number; we call these the *packet id*. The payload is assumed to remain unchanged during the course of the flooding (in some protocols, payloads do change as packets are routed; we discuss the handling of this kind of mutable payloads later in the paper).

To encode the payload of the data packets into the compensation packet, we use the XOR (operator $\otimes$), which is the simplest and best known FEC mechanism. A new data packet is added to the compensation packet by computing the XOR of its payload with the current payload in the compensation packet (initially, zero). Obviously, much more sophisticated error correction mechanisms are also possible; the advantage of XOR is its simplicity and low computational overhead.

Algorithm 1 Mistral's algorithm, code of node n_i.

```
1: Initialisation:
2:   DpBuffer ← ∅                                   {Received dps}
3:   cp ← ⊥                                         {Compensation packet}
4:   CpBuffer ← ∅                                   {Received cps}

5: upon flood(dp) do
6:   broadcast(dp)

7: upon reception of data packet dp for the first time do
8:   process(dp)
9:   runRecovery(dp)

10: upon reception of compensation packet cp from sender p_j
    do
11:   if cp.ids contains unknown dp ID then
12:     if recovery possible then
13:       dp_recov ← recover from cp
14:       process(dp_recov)
15:       runRecovery(dp_recov)
16:     else
17:       CpBuffer ← CpBuffer ∪ {cp}
18:       if level-2 recovery then
19:         expand(cp)
20:         for all recovered dp do
21:           process(dp)
22:           runRecovery(dp)

23: procedure process(dp)                  {handles a data packet}
24:   DpBuffer ← DpBuffer ∪ {dp}
25:   if F(dp) then
26:     broadcast(dp)
27:   else
28:     composeCompensationPac(dp)
29:   deliver dp to the application

30: procedure composeCompensationPac(dp)   {constructs a
    cp}
31:   cp.payload ← cp.payload ⊗ dp.payload
32:   cp.ids ← cp.ids ∪ {dp.id}; cp.ttls ← cp.ttls ∪ {dp.ttl}
33:   if |cp.ids| ≥ c then{|X| returns the nbr of elements in
      X}
34:     broadcast(cp)
35:     cp ← ⊥

36: procedure runRecovery(dp)              {recovers dps from
    CpBuffer}
37:   for all cp1 ∈ CpBuffer do
38:     if dp.id ∈ cp1.ids then
39:       remove dp from cp                {including TTL and ID}
40:       if recovery from cp1 possible then
41:         dp_recov ⟵ recover from cp1
42:   for all recovered data packets dp'_recov do
43:     process(dp'_recov)
44:     runRecovery(dp'_recov)

45: procedure expand(cp)                   {level-2 recovery}
46:   for all cp1 ∈ CpBuffer do
47:     for all cp2 ∈ CpBuffer ∧ cp2 ≠ cp1 do
48:       if cp1 or cp2 is reducible then
49:         cp ← reduction from cp1 and cp2
50:         CpBuffer ← CpBuffer ∪ {cp}
```

If the receiver of a compensation packet already has all but one of the contained data packets, the compensation packet will allow the reconstruction of that missing data packet. However, the recipient of a compensation packet has no a-priori way to know what data packets were used to build the compensation packet. Accordingly, compensation packets must include a list of all its contained data packet IDs. Assuming IP-style node addresses, the sender ID is represented by four bytes. The local sequence number consists of one byte, which allows Mistral to send 255 flooded messages by a node before looping back to 0. From this, we can see that the size of a compensation packets will be the payload size plus five times the number of included data packets c, i.e, $|cp| = |payload_{dp}| + 5 * c$. Notice that the packet size is independent of the number of nodes in the system as a whole. This information is sufficient for floodings that span the entire node field.

A complication arises in applications where the the scope of flooding is limited by a time-to-live (TTL) parameter. Here, the compensation packets need to represent the TTL for each contained data packet; otherwise, if a node recovers data packet dp from a compensation packet, it has no way to know what TTL to use when rebroadcasting dp. If it chooses a TTL that is smaller than the true TTL, then the flooding may die out too early. If the TTL is too high, then valuable bandwidth is wasted. Even worse, if the flooding is a part of a routing mechanisms and the routing mechanism depends on the TTL, then loops occur in the routing paths.

Clearly we cannot treat the TTL of a data packet as a part of that packet's payload, since TTLs are decremented at every hop of the data packet. The problem here is that incoming TTLs for received packets might differ at the node undertaking the reconstruction relative to the node that built the compensation packet. Thus, TTLs need to be added to the compensation packet outside of the payload.

The simplest approach is to add a list of TTLs to the compensation packet. Since the TTL is generally represented by one byte another c bytes are added to the size of a compensation packet. In effect, the TTL extends the packet-id by one byte.

Unfortunatly, this simple approach adds additional overhead, which we would like to avoid. A first point to notice is that TTLs are often defined based on some estimate and are thus, by design, already an approximation. Hence, if we manage to limit the error to some low number, we can manage with an approximate recnstruction of the TTL value. For instance, we could store the sum of all TTLs. The TTL of a recovered data packet can then be restored by subtracting the TTL's of all known packets (all data packets except one). To limit the size to one byte, we apply the modulo operator to this sum. Using this approach, the error will in most cases be within ± 1, or in total $\pm c$, which is acceptable for most applications. Thus, the total size of a compensation packet is $5c + 1 + |payload_{dp}|$ bytes.

Although we have not explored the idea yet, it may be possible to further reduce the overhead associated with compensation packets by compressing packet-id information. For example, in a MANET where most communication originates with a very small set of senders, we could assign those senders some sort of very small id. Moreover, it may sometimes be possible to compress the compensation packet payload itself. On the other hand, such ideas increase the computational overhead at the receiver and hence would require careful evaluation.

3.2.2 *Recovering from Compensation Packets*

To recover data packets from compensation packets we use a two-level recovery mechanism. The first level recovers data packets based on the data packets that have already been received. If $c - 1$ data packets contained in a compensation packet are known, the missing one can be reconstructed. Compensation packets that contain two or more missing data packets are stored (in the $CpBuffer$) and reconsidered when new data packets arrive or are recovered from other compensation packets. Actually, we do not store complete compensation packets, but only compensation packets that contain the IDs, TTL(s), and payload of the missing packets. More specifically, we xor the known data packet payloads with the payload of the compensation packet. After some time compensation packets are garbage collected, as it has become highly unlikely that the missing data packet(s) will be received in the future.

The level-2 recovery mechanism is more elaborate. Instead of only considering incoming and recovered data packets this algorithm also matches compensation packets against each other. The matching operation works as a reduction. Each new compensation packet is compared with all stored compensation packets. If either one of the packets is completely contained in the other, then a new compensation packet is added, which contains the set of data packet IDs of the larger packet minus the ones in the smaller packet. The new payload is constructed by applying XOR to both compensation packets. Provided that it does not allow the immediate recovery of a data packet, this reduced compensation packet is then added to the set of stored compensation packets (in $CpBuffer$).

Clearly, level-2 recovery adds a considerable overhead, both in storage and computation. Its application thus makes sense only if the gain in recovered data packets is significant with respect to level-1 recovery. We explore level-1 and level-2 recovery using simulations in Section 5.2.3.

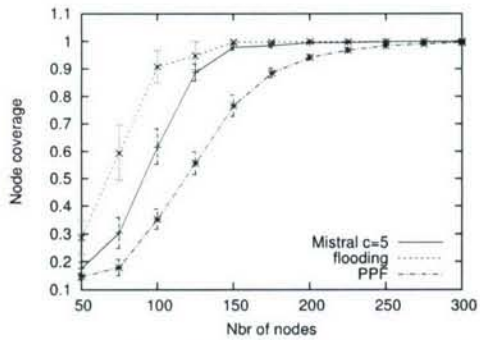
3.2.3 *Mutable Payloads*

Many routing protocols modify the flooded packets during the flooding. We have already shown how to handle TTL values. But some protocols modify other parts of data packets, for example by touching internal parameters, building a route trace, etc. To allow Mistral to handle these cases, we extend the above mechanism into compensation packets that include a mutable part and an immutable part of the payload. Clearly, the larger the immutable part is relative to the mutable part, the better the performance of Mistral. This is particularly the case as the immutable parts can be reduced into an immutable part of the same size, while mutable parts need to be appended to each other, thereby resulting in a size of $\sum_{i=0}^{c} mutablePartOf(dp_i)$. In genernal, the size of a compensation packet will now be $5c + |immutablePayload_{dp}| + \sum_{i=0}^{c} mutablePartOf(dp_i)$.

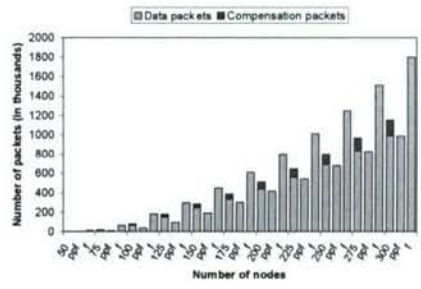
In the evaluation that follows, we assume that packets contain no mutable data other than the TTL.

4. ANALYSIS

In this section, we provide a simple analysis of Mistral. We denote by d_{max} the maximal diameter of the node field and consider floodings that span the entire node field. The maximal transmission latency $t_{maxTrans}$ is the maximal trans-



(a) Node coverage with varying density, $p = 0.55$.



(b) Message overhead with varying density, $p = 0.55$.

Figure 1: Node coverage and message overhead with varying node density.

mission range (88m) divided by the transmission speed. The time needed to do all the computations on a node is Δt, and we assume that there are no delays in the outgoing sending buffers, i.e., that there is no contention in the transmission medium.

Let f_i denote the number of floodings originating at node i, then the estimated overall generated number of compensation packets in a network with n nodes is $G = n\frac{(1-p)\Sigma_i f_i}{c}$, assuming that every node receives all flooded data packet at least once. Thus, the overhead in bytes is $G * (5c + 1 + |payload_{dp}|)$.

Assume that δ_{flood} denotes the average reception frequency of data packet that are received for the first time. Then, the estimated time needed to fill up a c-based compensation packet is $t_{recoveryPac} = \frac{c}{(1-p)*\delta_{flood}}$.

We now consider the delivery latency of a data packet. The worst case occurs when the flooding source and the destination are d_{max} hops apart and the data packet is always forwarded as part of a compensation packet. In this case, the maximum delivery latency is $d_{max} * (t_{recoveryPac} + \Delta t + t_{maxTrans})$, while the estimated maximum delivery latency is $(1-p) * d_{max} * (t_{recoveryPac} + \Delta t + t_{maxTrans}) + p * d_{max} * (\Delta t + t_{maxTrans})$.

We now compute the number of packets sent by a single flooding in a network of N connected nodes. Purely probabilistic flooding has a message overhead of $E(MsgOverhead) = p * N$, if we assume that every node receives the flooded message at least once. Mistral adds an estimate of $\frac{1}{c}$ for every dropped message. Thus, the total overhead per flooding is $(1-p) * N * \frac{1}{c} + p * N$. If the assumption that all nodes receive the flooded message is relaxed then the relative overhead added by Mistral increases. Each node that receives the flooded message only because of Mistral again contributes an additional broadcast or partial compensation to the overhead. Naturally, the additional overhead pays off through the increased node coverage.

5. SIMULATIONS

For our simulation we used JiST/SWANS v1.0.4 [1, 4], a simulation environment for ad hoc networks. Java applications written for a real deployment can be ported to the simulation environment and then placed under a variety of simulated scenarios and loads. JiST/SWANS intercepts the calls to the communication layer and dynamically transforms them into calls to the simulator's communication package.

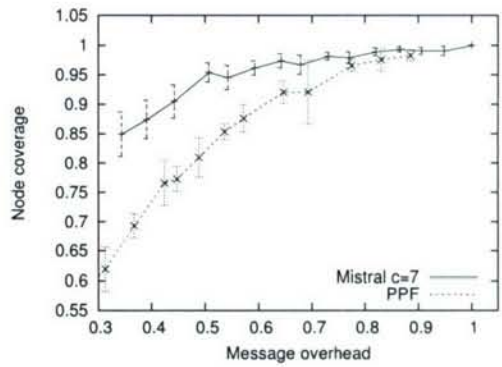
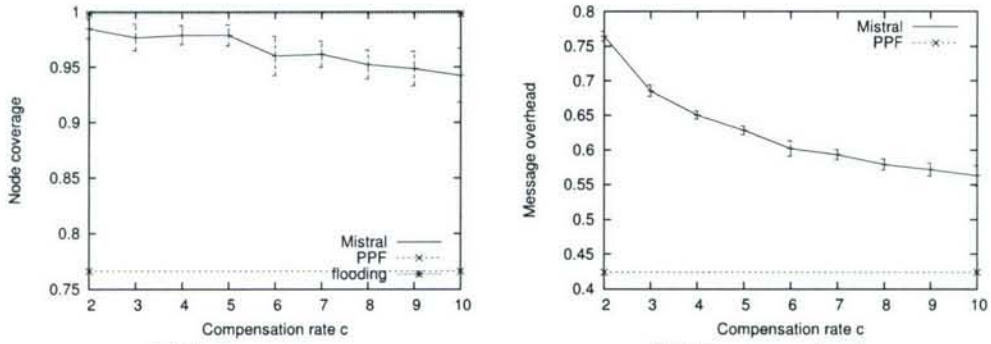


Figure 2: Node coverage with respect to message overhead.

5.1 Setup

We consider a set of nodes. Communication between two nodes m and n occurs in an ad hoc manner and may be asymmetric, i.e., n may be able to communicate with m, but the inverse may not be possible. Communication is by broadcast as defined in the 802.11b standard [12] and can be subject to interference, in which case the message cannot be received. Interference can occur without the sender being able to detect the problem (this is called the *hidden terminal problem* [2]).

We simulate a wireless ad-hoc network with 150 nodes uniformly distributed in a field of size 600x600m. Nodes are stationary, except for one case in which we measure the impact of mobility (Section 5.2.4). The maximal transmission range of a node is set to 88m. Every node starts flooding 20 messages at a regular interval, once all nodes are started up. All flooding occurs across the entire node field. Hence, ideally all nodes should receive all flooded messages.



(a) Node coverage. (b) Message overhead.

Figure 3: Varying compensation rate c, $p = 0.55$.

Our work models disconnections due to mobility, transmission range limits, and the hidden terminal problem just mentioned (using JiST/SWANS' *RadioNoiseIndep* package, which uses a radio model identical to ns2). Unless otherwise mentioned, we use the default values defined in JiST/SWANS.

The nodes start up at random times and positions. When they are all up and running, we start sending the flooding messages and we wait until all messages have been received (terminating simulation).

5.2 Results

In this section, we present the results of our simulation. Every node periodically, every 50s, floods a message throughout the entire field. We have chosen a low flooding rate because in our simulations we want to minimize the effect of packet loss due to buffer overflows and interference. The nodes are added to the sensor field at time 0s but start flooding at times uniformly distributed between 0 and 60s. All results give the average over at least 30 runs in different uniform node distributions. In general, the variance in the simulation results for ad hoc networks is high. This is due to the many sources of randomness: distribution of the sensor nodes, the paths of nodes, the time the nodes flood a message, etc. Thus, where significant we indicate the 95%-confidence intervals (CI).

To evaluate the quality of Mistral, we are mainly interested in two properties: node coverage and message overhead. *Node coverage* measures the number of nodes that have received the messages, while message overhead indicates the total number of sent messages. Both measurements are normalized against a connected network with the same number of nodes. In a connected network, any node can communicate with any other node. Thus, node coverage is given as a percentage of all nodes in the network, while message overhead is given as the percentage of the message overhead in the case in which all nodes receive all messages (normal flooding). Note that the message overhead in the connected network equals the product of the number of flooded messages with the number of nodes. Moreover, it is generally lower in a network with partitions. Since Mistral complements local-knowledge-based approaches and is based on purely probabilistic flooding, we compare Mistral to the latter. Purely probabilistic flooding is entirely defined by the rebroadcasting probability p. For completeness, we also show the results for simple flooding, which corresponds to PPF with $p = 1.0$.

In the following, we evaluate the following properties of Mistral: its behavior in the face of varying density, varying protocol parameters, node mobility, packet loss, and with the secondary recovery mechanism. Unless explicitly stated otherwise, we use the above default values in our measurements.

5.2.1 *Impact of Density*

We start by measuring the impact of node density on the node coverage and the message overhead. Fig. 1(a) shows the node coverage with varying number of nodes. It shows three measurements: simple flooding, purely probabilistic flooding (PPF), and Mistral with compensation rate $c = 5$. The rebroadcast probability is set to $p = 0.55$ in the cases of purely probabilistic flooding and Mistral. As expected, Mistral has a much higher node coverage than purely probabilistic flooding, especially for lower node densities. If the node density passes a certain threshold (around 225 nodes for Mistral), it is sufficiently high such that all nodes receive all messages. In contrast, with low density only a low percentage of the nodes receive all messages. However, below a certain threshold (around 150 nodes) even simple flooding cannot reach all nodes.

In Fig. 1(b) we show the corresponding message overhead. For every number of nodes indicated on the x-axis, we draw the sent number of packets for Mistral, purely probabilistic flooding (ppf), and simple flooding (f). Mistral's packets are further separated into data packets and compensation packets. Since Mistral adds additional compensation packets, its total message overhead is higher than the one of purely probabilistic flooding. Notice also that for low densities the number of flooding packets is higher. Due to higher node coverage in Mistral, more nodes receive the message and thus more nodes also rebroadcast the message, which accounts for the higher number of flooding packets compared to PPF.

Thus, to measure Mistral's net gain in node coverage, as compared to purely probabilistic flooding, we need to consider both node coverage and message overhead graphs. Indeed, since Mistral's compensation mechanism adds an additional overhead, we cannot directly compare the two approaches with the same rebroadcast probability p. Rather,

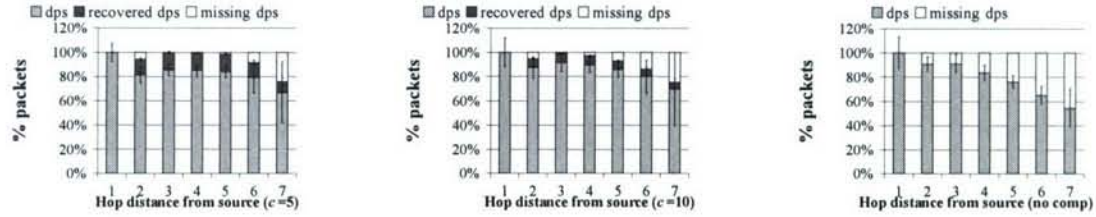


Figure 4: Recovery based on hop counts, single source and $p = 0.55$.

we need to compare Mistral with the purely probabilistic flooding using a rebroadcasting probability with a similar message overhead. Fig. 2 plots the node coverage with respect to the message overhead, for $c = 7$. The message overhead corresponds to simulation runs with p varying from 0.3 (0.4 for PPF) to 1, in steps of 0.05. The gain with Mistral is especially prominent for low rebroadcast probabilities p. Of course, low rebroadcast probabilities lead to many dropped rebroadcasts and thus the node coverage becomes low. Using Mistral allows some of the nodes to recover messages they may have missed. For an overhead of 0.35, Mistral improves the node coverage by 20%, for an overhead of 0.55 by 10%, and for overhead around 0.75 it is closer to 3%.

5.2.2 *Compensation Rate*

We now turn to one of the parameters that determine the behavior of Mistral: compensation rate c. In Fig. 3(a), we show the node coverage with compensation rate c varying from 2 to 10, for 150 nodes and rebroadcasting probability $p = 0.55$. Generally, the node coverage decreases with increasing compensation rate. For comparison, the graph also indicates the node coverage for flooding and purely probabilitstic flooding (PPF) with the same parameters. Both flooding and probabilistic flooding are independent from the compensation rate and thus are represented by a horizontal line. Fig. 3(b) gives the corresponding message overhead. Here, the message overhead decreases with increasing compensation rate. Thus, given a particular node coverage the higher the compensation rate the better. However, a higher compensation rate also increases the message delivery latency. Indeed, data packets that are part of a compensation packet spend more time waiting until the compensation packet is filled with sufficient data packets and may thus be delayed.

5.2.3 *Recovery Performance and Overhead*

Next, we measure the number of recovered data packets with respect to the hop count (see Fig.4). In this simulation, a single node at position $[300, 300]$ periodically floods 1000 messages. We give the results for $c = 5$, $c = 10$, and the case with no compensation (no comp). Since the overall number of received data packets is different depending on the hop-distance of a node to the flooding source, we give the percentage of recovered data packets to all flooding packets that should have been received by the nodes at this hop distance from the source. The percentage of recovered data packets is approximately the same for most hop distances. An exception is at hop count 1, where all nodes generally receive the flooded message, because the source floods the data packet with $p = 1.0$ and at a time of low traffic. As fewer compensation packets are sent in the case of $c = 10$, the percentage of recovered data packets is lower compared to the case of $c = 5$. Towards very high numbers of hop counts, no compensation packets are received. However, these nodes are particular cases resulting from a unusual node distribution, which does not occur frequently.

Notice that the percentage of dps increases between $c = 5$ and $c = 10$. The reason is that with smaller c, more compensation packets are sent and the likelihood that a dps is received via a compensation packet increases. Since we count the data packets when they are received for the first time, more packets are received via a compensation packet. Thus, the percentage of dps is smaller for a smaller c.

We use the same setup to measure the packet delivery latency. In contrast to the other simulations, we use a single data point (one random uniform distribution) in this case. The single source floods a data packet every second. The graphs in Fig. 5(a) and (b) show the latency distribution of data packets for $c = 2$ and $c = 8$ with respect to the hop distance of the node. The delivery latency of a data packet is high if it is received by a node only as part of a compensation packet. The higher the compensation rate, the higher this delay is.

Another important characteristic of Mistral is the ratio of compensation packets that cannot be recovered. We say that a *compensation packet is recovered* if all contained data packets have been received or have been recovered. In general, we expect the number of unrecovered compensation packets to increase with increasing compensation rate. The graph in Fig. 5(c) confirms this. It uses our default setup with many flooding sources and shows the total number of received compensation packets and the number of compensation packets that have not been recovered (logarithmic scale on y-axis). Clearly, the lower the compensation rate, the higher the number of sent and thus received compensation packets. This number also includes all compensation packets whose contained data packets have already been received earlier by the receiving node (useless cps). Immediately recovered compensation packets denote the compensation packets that only contain a single unknown data packet. Any compensation packet that contains more unknown data packets is added to *CpBuffer*.

Fig. 5(d) shows the number of sent compensation packets based on the number of nodes in the field. As expected, this